

Seed Specification Language Documentation

Contents

1 Seed Specification Language	8
Quick Example	8
□ Key Features	9
□ Type System	9
□ Generated Stack	10
Frontend	10
Backend	10
Infrastructure	10
□ Documentation	10
□ Development Status	11
□ Contributing	11
□ License	11
□ Contact & Support	11
□ Acknowledgments	12
2 .aider.chat.history.md	13
3 aider chat started at 2024-12-17 21:29:06	14
4 aider chat started at 2024-12-17 22:32:43	21
5 CONTRIBUTING.md	41
6 Contributing to SeedML	42
Getting Started	42
Development Setup	42
CLI Development	42
Contribution Areas	43
Code Style	43
Pull Request Process	43
Questions?	43
License	44
7 LICENSE-GPL.md	45
8 docs/getting-started.md	46
9 Getting Started with SeedML	47
Installation	47
Quick Start	47
Next Steps	47
Learning Path	48
10 docs/getting-started/basic-concepts.md	49

11 Basic Concepts	50
Core Structure	50
Type System	51
1. Basic Types	51
2. UI-Specific Types	51
Modules and Components	52
1. Module Structure	52
2. Component Schema	53
Best Practices	53
12 docs/getting-started/first-app.md	55
13 Creating Your First App	56
1. Basic Structure	56
2. Add Features	56
3. Add Business Rules	57
Next Steps	58
14 docs/getting-started/installation.md	60
15 Installing Seed Spec	61
Prerequisites	61
Installation	61
Configuration	61
Verify Installation	61
Current Status	61
Next Steps	62
16 docs/getting-started/introduction.md	63
17 Introduction to SeedSpec	64
Why SeedSpec?	64
Key Features	64
1. Type Safety First	64
2. Clear Module System	64
3. Component Schemas	65
4. Design System Integration	65
Benefits	65
Getting Started	66
18 docs/getting-started/quick-start.md	67
19 Quick Start Guide	68
1. Define Your App	68
2. Run the Generator	69
3. Start the App	69
Next Steps	69
20 docs/core-concepts/architecture.md	71
21 Architecture & Design	72
System Architecture	72
Layer Dependencies	72
Core Components	73
1. Parser & Validator	73
2. Generation Engine	74
3. Output Processor	74

4. Location Services	75
Design Principles	75
1. Separation of Concerns	75
2. Reliability	75
3. Extensibility	75
Implementation Details	75
1. Validation Pipeline	75
2. Generation Process	76
3. File Management	76
Error Handling	77
Cross-Compilation Architecture	77
Cross-Compiler Components	77
Cross-Compilation Process	78
Future Directions	78
1. Near-term Improvements	78
2. UI Patterns & Components	78
3. Research Areas	79
3. Tooling	79
Contributing	80
Core Concepts	80
Key Features	80
1. Application Patterns	80
2. Scalability	81
3. Resilience	81
4. Distribution	81
Best Practices	81
22 docs/core-concepts/business-rules.md	83
23 Business Rules	84
Core Concepts	84
1. Basic Validation	84
2. Simple State Transitions	84
3. Basic Computations	85
Common Patterns	85
1. Field Validation	85
2. Cross-field Validation	85
3. Simple Workflows	85
Best Practices	86
1. Keep Validations Simple	86
2. Use Clear State Transitions	86
3. Minimize Complexity	86
4. Prefer Convention	86
Key Benefits	87
24 docs/core-concepts/core-qualities.md	88
25 Core Qualities	89
1. AI-First Design	89
2. Intent Over Implementation	89
3. Single Source of Truth	90
4. Smart Defaults	90
Why These Matter	90
Core Concepts	91
Key Features	91

1. Performance	91
2. Reliability	91
3. Maintainability	92
4. Testability	92
Best Practices	92
26 docs/core-concepts/integration.md	94
27 Integration	95
Core Integration Patterns	95
1. Smart Authentication	95
2. Data Integration	95
3. Service Communication	95
4. Location Services	96
5. Event Handling	96
6. Maps Integration	97
Best Practices	97
1. Role-Based Access	97
28 docs/core-concepts/language-structure.md	99
29 Seed Specification Language Structure	100
Core Principles	100
1. Type Safety First	100
2. Modular Design	100
3. Component-Based Structure	101
Key Components	101
1. Type System	101
2. Design System	102
3. Components	103
Best Practices	104
30 docs/core-concepts/modular-apps.md	105
31 Modular App Specifications	106
Basic Usage	106
Key Benefits	106
File Organization	107
Loading Order	107
Validation	107
Best Practices	108
32 docs/core-concepts/overview.md	109
33 Core Concepts	110
Modular Apps	110
Key Ideas	110
Core Features	110
Learn More	111
34 docs/core-concepts/security.md	112
35 Security	113
Core Security Features	113
1. Authentication	113
2. Authorization	114
3. Data Protection	114

4. Audit Logging	114
Security Patterns	115
1. Data-Level Security	115
2. User Management	116
3. Location Privacy	116
36 docs/core-concepts/smart-defaults.md	117
37 Smart Defaults	118
Core Principles	118
1. Convention Over Configuration	118
2. Progressive Complexity	118
3. Contextual Awareness	119
Common Default Patterns	119
1. Type-Based Defaults	119
2. UI Patterns	119
3. Business Logic	120
4. Security	121
Overriding Defaults	121
Benefits	121
38 docs/core-concepts/theming.md	123
39 Component-Based Theming	124
Basic Usage	124
Component Tokens	125
Button Variants	125
Form Elements	125
Typography	125
Dark Mode	126
Theme Composition	126
Smart Defaults	126
40 docs/core-concepts/type-system.md	127
41 Type System	128
Core Types	128
UI-Specific Types	129
Type Usage	129
1. Explicit Type Declaration	129
2. Component Type Validation	130
3. Entity Type Validation	130
Type Composition	131
Benefits	131
42 docs/patterns/advanced-patterns.md	133
43 Advanced SeedML Patterns	134
Multi-Tenant Architecture	134
Complex Workflows	134
Plugin Architecture	135
Event Sourcing	135
AI Integration	136
Time-Based Patterns	136
Best Practices	137
44 docs/reference/cli.md	138

45 Command Line Interface	139
Quick Start	139
Essential Commands	139
Location Commands	139
Maps Configuration	139
Configuration	140
Generated Stack	140
Learn More	140
46 docs/reference/patterns.md	141
47 Application Patterns	142
Foundation Layer Patterns	142
1. Foundation Patterns	143
2. Data Patterns	143
3. Logic Patterns	143
4. Security Patterns	143
5. Presentation Patterns	144
6. Integration Patterns	144
CRUD Operations	144
Workflow Management	144
Dashboard Layouts	145
Form Handling	145
Search and Filter	145
Access Control	146
Integration Patterns	146
Multi-tenant Patterns	146
Best Practices	147
48 docs/reference/types.md	148
49 Type System Reference	149
Core Types	149
Basic Validation	149
Type Inference	150
Custom Types	150
Type Composition	151
Best Practices	151
Basic Structure	151
Entity Syntax	152
50 docs/examples/basic-crud.md	153
51 Basic CRUD Example	154
What You Get	155
Data Layer	155
User Interface	155
Features	155
Map Features	155
Progressive Enhancement	156
Key Principles Demonstrated	156
52 docs/examples/business-app.md	158
53 Business Application Example	159
54 docs/examples/dashboard.md	168

55 Analytics Dashboard Example	169
56 docs/examples/integration.md	172
57 Integration Examples	173
Basic Integration	173
Common Patterns	174
1. Authentication	174
2. File Storage	174
3. Email Service	174
Best Practices	174
Smart Defaults	174
58 docs/examples/saas.md	175
59 SaaS Application Example	176
60 .pytest_cache/README.md	179
61 pytest cache directory	180

Chapter 1

Seed Specification Language

Project Status Language Design license Dual GPL/Commercial docs latest

SeedSpec is a token-efficient, declarative language optimized for LLM generation that produces full-stack business applications from clear specifications. Write what you want with strong typing, get working software.

Quick Example

```
mod todo {
  use { @core, @ui }

  // Domain model
  data {
    Task {
      title str(3..100)
      done bool
      due @future
    }
  }

  // Business process
  flow TaskFlow {
    new -> active: assign
    active -> done: complete
    * -> archived: archive
  }

  // UI components
  comp task_card {
    in { task Task, done fn }
    style {
      bg white
      pad med
      border @primary
    }
  }
}
```

```

// Screens
ui {
  TaskList {
    layout grid(3)
    show task_card
    acts { create, done }
    rules {
      create needs title
      done when all done
    }
  }
}

// Theme
theme {
  colors {
    primary #0066cc
    success @green.5
    error @red.5
  }
  space {
    sm 4
    med 8
  }
}
}

```

□ Key Features

- **Token Efficient:** Optimized syntax for LLM generation while maintaining readability
- **Pure Declarative:** Clear separation of what vs how across all features
- **Full Stack:** Complete coverage from UI to database, workflows, and agents
- **Smart Defaults:** Production patterns built-in with context-aware inference
- **Type Safety:** Catch errors at compile time with explicit types and validation
- **Tech Independent:** Target any modern technology stack
- **Standard Library:** Rich set of pre-built components, themes, and patterns

□ Type System

SeedSpec uses a token-efficient type system with smart inference:

```

// Core types
type str(min?..max?) {
  match?: regex
  format?: @email|@url
}

type num(min?..max?) {
  int?: bool
  pos?: bool
}

// UI types

```

```

type color = hex|rgb|@token
type size = px|rem|@token

// Components
comp btn {
  in { txt str, click fn }
  opt { disabled bool }
  var { pri, sec }
}

```

□ Generated Stack

SeedSpec generates a complete, type-safe stack:

Frontend

- React + TypeScript
- Type-safe components
- Strongly typed state management
- Validated forms
- Type-safe API integration

Backend

- FastAPI + SQLAlchemy
- Type-safe endpoints
- Schema validation
- Error handling
- Database type safety

Infrastructure

- Type-safe migrations
- Configuration validation
- API type definitions
- Test type coverage

□ Documentation

- **Getting Started**
 - [Introduction](#)
 - [Installation](#)
 - [Quick Start](#)
 - [First Application](#)
- **Core Concepts**
 - [Type System](#)
 - [Language Structure](#)
 - [Business Rules](#)
 - [Architecture](#)
- **Examples**
 - [Basic CRUD](#)
 - [Business App](#)
 - [Dashboard](#)

- [SaaS](#)

□ Development Status

SeedSpec is in active development (v0.1.0) with a focus on:

1. **Type System**
 - Strong type checking
 - Compile-time validation
 - Clear error messages
 - IDE integration
2. **Module System**
 - Explicit imports/exports
 - Clear boundaries
 - Dependency management
 - Version control
3. **Developer Experience**
 - Type-aware IDE support
 - Real-time validation
 - Debug tools
 - Error tracing
4. **Enterprise Features**
 - Type-safe multi-tenancy
 - Authentication schemas
 - Authorization rules
 - Audit logging
 - Compliance validation

□ Contributing

We welcome contributions! See [CONTRIBUTING.md](#) for: - Development setup - Coding standards - Pull request process
- Issue guidelines

□ License

- **Open Source:** [GNU GPL v3.0](#)
 - Free for personal and open source use
 - Modifications must be shared
 - Commercial use requires license
- **Commercial License**
 - Coming soon
 - Priority support
 - Private modifications
 - Enterprise features

□ Contact & Support

- **Website:** <https://noodleseed.com>
- **Documentation:** <https://docs.noodleseed.com>
- **GitHub Issues:** Bug reports & feature requests
- **Discord:** Community chat & support
- **Email:** info@noodleseed.com

- **Twitter:** [[@noodleseed](https://twitter.com/noodleseed)](<https://twitter.com/noodleseed>)

□ **Acknowledgments**

Special thanks to: - Anthropic for Claude API access - Our open source contributors - Early adopters and testers

Built with ♥ by Noodle Seed Making software development more type-safe and maintainable

Chapter 2

.aider.chat.history.md

Chapter 3

aider chat started at 2024-12-17 21:29:06

```
/home/codespace/.python/current/bin/aider
Aider v0.69.1
Main model: claude-3-5-sonnet-20241022 with diff edit format, infinite output
Weak model: claude-3-5-haiku-20241022
Git repo: .git with 69 files
Repo-map: using 1024 tokens, auto refresh
https://aider.chat/HISTORY.html#release-notes
Would you like to see what's new in this version? (Y)es/(N)o [Yes]: n
```

give me a summary of this repo, all src code and everything related to themes and examples and tools.

Based on the repository structure, this appears to be a theming/styling framework with the following key components:

1. Core Theme System:
 - Located in `src/theme/` with `ThemeContext.tsx` and `theme.css`
 - Standard theme definitions in `src/stdlib/themes/` (`base.seed`, `dark.seed`, `light.seed`)
2. Theme Processing Tools:
 - `tools/compile_react_theme.py`: Main compiler that:
 - Generates CSS variables from theme specs
 - Creates theme context for React
 - Handles theme overrides and predefined themes
 - Flattens theme objects
3. Parser:
 - `src/parser.py`: Handles parsing of `.seed` files which appear to be the theme definition format
 - Includes error handling via `SeedParseError`
 - Supports file imports and relative paths
4. Example Applications:
 - `examples/react-theme/`: Complete React application showing theme usage
 - Uses Vite as build tool
 - Includes `ThemeContext` implementation
 - Shows practical theme application
 - `examples/app-theme.seed` and `custom-theme.seed`: Theme definition examples
5. Testing:
 - Comprehensive test suite under `tests/`

- Unit tests for imports, parsing, React theme compilation
- Integration tests
- Test fixtures with various .seed files showing different use cases

6. Documentation:

- Extensive markdown docs covering:
 - Core concepts (architecture, theming, type system)
 - Getting started guides
 - Examples and patterns
 - Reference documentation

The system appears to be designed to: 1. Define themes using a custom .seed format 2. Parse and validate these definitions 3. Compile them into CSS variables and React context 4. Provide tools for theme manipulation and override 5. Support modular theme composition through imports

Would you like me to examine any specific component in more detail?

Tokens: 3.0k sent, 362 received. Cost: \$0.01 message, \$0.01 session.

Add examples/app-theme.seed to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: y

Add examples/custom-theme.seed to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add src/parser.py to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add src/parser.py to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add src/stdlib/themes/dark.seed to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add src/stdlib/themes/dark.seed to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add tools/compile_react_theme.py to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add tools/compile_react_theme.py to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Based on the files provided, this appears to be a theming system with the following components:

1. A custom theme definition language called "Seed" (.seed files):
 - Custom theme definitions (examples/custom-theme.seed)
 - App theme configurations (examples/app-theme.seed)
 - Base themes in stdlib (dark.seed, and references to light.seed and base.seed)
2. Parser Infrastructure:
 - src/parser.py - Parses .seed files into Python dictionaries
 - Handles imports, nested blocks, and various value types
 - Prevents circular imports
3. Theme Compilation Tool:
 - tools/compile_react_theme.py - Converts Seed themes to React
 - Generates CSS variables
 - Creates React context/provider components
 - Handles theme overrides and inheritance

To get a complete picture, we should also look at:

1. Base theme definitions:

```
src/stdlib/themes/base.seed
src/stdlib/themes/light.seed
```

2. React theme implementation:

src/theme/ThemeContext.tsx
src/theme/theme.css

3. Any tests:

tests/unit/test_theme_spec.py
tests/unit/test_react_theme_compiler.py

Would you like me to analyze any of these additional files? Please add them to the chat if you'd like me to propose any changes or improvements to the theming system.

The current implementation supports: - Custom theme definitions - Theme inheritance/extension - App-specific theme overrides - React integration - CSS variable generation

But we might find opportunities for improvement in areas like: - Theme validation - Default fallbacks - Runtime theme switching - Performance optimization - Additional framework support

Let me know which aspects you'd like to explore further!

Tokens: 5.8k sent, 346 received. Cost: \$0.02 message, \$0.04 session.

Add src/stdlib/themes/base.seed to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add src/stdlib/themes/light.seed to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add src/stdlib/themes/light.seed to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add src/theme/ThemeContext.tsx to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add src/theme/ThemeContext.tsx to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add src/theme/theme.css to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add src/theme/theme.css to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add tests/unit/test_react_theme_compiler.py to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add tests/unit/test_react_theme_compiler.py to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add tests/unit/test_theme_spec.py to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add tests/unit/test_theme_spec.py to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Let me provide a summary of the theming system in this repository:

Core Components: 1. Theme Definition Language (.seed files): - Uses a custom DSL for defining themes - Located in src/stdlib/themes/ - Has base.seed (default theme), light.seed, and dark.seed variants - Supports inheritance (extends keyword) - Defines colors, typography, spacing, borders, shadows

2. Parser (src/parser.py):

- Parses .seed files into Python dictionaries
- Handles imports, comments, and different value types
- Supports circular import detection
- Error handling with SeedParseError

3. React Theme Compiler (tools/compile_react_theme.py):

- Converts .seed theme definitions to React components
- Generates:
 - CSS variables

- TypeScript theme context
- Supports theme overrides
- CLI tool with customizable input/output paths

4. React Integration:

- ThemeContext.tsx for React context
- theme.css for CSS variables
- Theme Provider component
- useTheme hook for consuming theme values

5. Examples:

- app-theme.seed: Basic theme usage
- custom-theme.seed: Advanced theme customization
- React theme example showing implementation

6. Tests:

- test_theme_spec.py: Theme parsing tests
- test_react_theme_compiler.py: Compiler tests

The system allows for: - Hierarchical theme definitions - Theme inheritance - Runtime theme overrides - Type-safe theme usage in React - CSS variable generation

All key files appear to be present in the chat. The structure is complete for the theming functionality.

Would you like me to propose any specific improvements or changes to this system?

Tokens: 8.0k sent, 332 received. Cost: \$0.03 message, \$0.07 session.

/chat-mode ask

Aider v0.69.1
 Main model: claude-3-5-sonnet-20241022 with ask edit format, infinite output
 Weak model: claude-3-5-haiku-20241022
 Git repo: .git with 69 files
 Repo-map: using 1024 tokens, auto refresh
 Added examples/app-theme.seed to the chat.
 Added examples/custom-theme.seed to the chat.
 Added src/parser.py to the chat.
 Added src/stdlib/themes/base.seed to the chat.
 Added src/stdlib/themes/dark.seed to the chat.
 Added src/stdlib/themes/light.seed to the chat.
 Added src/theme/ThemeContext.tsx to the chat.
 Added src/theme/theme.css to the chat.
 Added tests/unit/test_react_theme_compiler.py to the chat.
 Added tests/unit/test_theme_spec.py to the chat.
 Added tools/compile_react_theme.py to the chat.

show me an example of defining a theme, compiling it and using it in a react app

I'll show you a complete example from start to finish:

1. First, let's create a custom theme file my-app-theme.seed:

```
app MyApp {
  ui {
    theme: "light" // Extend the light theme

    overrides: {
```

```

    colors.primary: "#0055ff"
    colors.secondary: "#6b2fbc"
    typography.fontFamily.base: "Poppins, sans-serif"
    typography.fontSize.base: "16px"
    spacing.md: "1.25rem"
    borders.radius.md: "8px"
  }
}
}

```

2. Compile the theme using the provided tool:

```
python tools/compile_react_theme.py my-app-theme.seed --output src/theme
```

This will generate two files: - src/theme/theme.css (with CSS variables) - src/theme/ThemeContext.tsx (with React context)

3. Create a React app that uses the theme:

```

// App.tsx
import { ThemeProvider } from "../theme/ThemeContext";
import "../theme/theme.css";

function Button({ children }) {
  return (
    <button
      style={{
        backgroundColor: "var(--colors-primary)",
        color: "var(--colors-background)",
        padding: "var(--spacing-md)",
        borderRadius: "var(--borders-radius-md)",
        fontFamily: "var(--typography-fontFamily-base)",
        fontSize: "var(--typography-fontSize-base)",
        border: "none",
        cursor: "pointer"
      }}
    >
      {children}
    </button>
  );
}

// Using the theme with the useTheme hook
import { useTheme } from "../theme/ThemeContext";

function ThemedComponent() {
  const theme = useTheme();

  return (
    <div style={{ color: theme.colors.text }}>
      <h1>Themed Component</h1>
      <p>This text uses the theme's text color</p>
    </div>
  );
}

```

```
// Main App
function App() {
  return (
    <ThemeProvider>
      <div>
        <h1>My Themed App</h1>
        <Button>Click Me</Button>
        <ThemedComponent />
      </div>
    </ThemeProvider>
  );
}
```

```
export default App;
```

4. You can also create a styled-components or emotion implementation:

```
// With styled-components
import styled from 'styled-components';

const StyledButton = styled.button`
  background-color: var(--colors-primary);
  color: var(--colors-background);
  padding: var(--spacing-md);
  border-radius: var(--borders-radius-md);
  font-family: var(--typography-fontFamily-base);
  font-size: var(--typography-fontSize-base);
  border: none;
  cursor: pointer;

  &:hover {
    background-color: var(--colors-secondary);
  }
`;
```

5. To switch themes dynamically, you could modify the ThemeProvider:

```
// DynamicThemeProvider.tsx
import { useState } from 'react';
import { ThemeProvider } from '../theme/ThemeContext';

const themes = {
  light: lightTheme,
  dark: darkTheme,
};

function DynamicThemeProvider({ children }) {
  const [currentTheme, setCurrentTheme] = useState('light');

  return (
    <ThemeProvider theme={themes[currentTheme]}>
      <button onClick={() => setCurrentTheme(current =>
        current === 'light' ? 'dark' : 'light'
      )}>
        Toggle Theme
      </button>
    </ThemeProvider>
  );
}
```

```
        {children}
      </ThemeProvider>
    );
  }
}
```

The benefits of this approach: 1. Type-safe theme usage with TypeScript 2. CSS variables for better performance 3. Centralized theme definition 4. Easy theme switching 5. Support for theme inheritance and overrides 6. Works with any styling solution (CSS-in-JS, CSS Modules, plain CSS)

Would you like me to explain any part in more detail?

Tokens: 6.3k sent, 821 received. Cost: \$0.03 message, \$0.10 session.

Chapter 4

aider chat started at 2024-12-17 22:32:43

```
/home/codespace/.python/current/bin/aider
Aider v0.69.1
Main model: claude-3-5-sonnet-20241022 with diff edit format, infinite output
Weak model: claude-3-5-haiku-20241022
Git repo: .git with 69 files
Repo-map: using 1024 tokens, auto refresh
```

/chat-mode ask

```
Aider v0.69.1
Main model: claude-3-5-sonnet-20241022 with ask edit format, infinite output
Weak model: claude-3-5-haiku-20241022
Git repo: .git with 69 files
Repo-map: using 1024 tokens, auto refresh
```

tell me about this repo and everything to do with examples, tools and theming system

From analyzing the repository structure, this appears to be a framework or library that has a strong focus on theming capabilities, particularly with React integration. Let me break down the key areas:

Theming System: - There appears to be a core theming system with standard themes in `src/stdlib/themes/` (base, dark, and light variants) - Theme-related React components in `src/theme/ThemeContext.tsx` - Theme-specific CSS in `src/theme/theme.css`

Examples: 1. React Theme Example: - A complete example in `examples/react-theme/` showing theme implementation with React - Includes standard React+Vite setup (`vite.config.ts`, `package.json`) - Demonstrates theme context usage (`ThemeContext.tsx`)

2. Seed Examples:

- `app-theme.seed`
- `custom-theme.seed` These appear to be template or configuration files for theming

Tools: The main tool visible is `tools/compile_react_theme.py` which has several key functions: - `generate_css()`: Converts theme specifications into CSS variables - `generate_theme_context()`: Likely creates React context code for themes - `load_predefined_theme()`: Loads built-in themes - `apply_theme_overrides()`: Handles theme customization - `extract_theme_from_app_spec()`: Pulls theme data from application specifications

Documentation: There's extensive documentation about theming in: - `docs/core-concepts/theming.md` - Various examples in the docs showing theme usage

Testing: Theme-related tests are present in: - `tests/unit/test_react_theme_compiler.py` - `tests/unit/test_theme_spec.py`

The system appears to use a custom “.seed” file format (parsed by `src/parser.py`) which might be a configuration or specification format for themes and applications.

Would you like me to dive deeper into any particular aspect? To provide more detailed information, I might need to see the contents of some key files.

Tokens: 823 sent, 401 received. Cost: \$0.0085 message, \$0.0085 session.

Add `docs/core-concepts/theming.md` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add `examples/app-theme.seed` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add `examples/app-theme.seed` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add `examples/custom-theme.seed` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add `examples/custom-theme.seed` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add `src/theme/ThemeContext.tsx` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add `src/theme/ThemeContext.tsx` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add `src/theme/theme.css` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add `src/theme/theme.css` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add `tests/unit/test_react_theme_compiler.py` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add `tests/unit/test_react_theme_compiler.py` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add `tests/unit/test_theme_spec.py` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add `tests/unit/test_theme_spec.py` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add `tools/compile_react_theme.py` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add `tools/compile_react_theme.py` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

This repository appears to contain a theming system implementation with several key components:

1. Theme Definition System:

- Uses a custom “Seed” specification format (.seed files) to define themes
- Supports a hierarchical theme system with inheritance (“extends”)
- Has built-in themes (light, dark, etc.) in the standard library
- Allows custom theme definitions and overrides

2. Theme Properties: The theming system covers:

- Colors (primary, secondary, background, etc.)
- Typography (font families, sizes, weights, line heights)
- Spacing scales
- Borders and shadows
- Smart defaults for production-ready styling

3. Tools and Compilation:

- `compile_react_theme.py`: A tool that:
 - Converts Seed theme specs to CSS variables
 - Generates React context for theme consumption

- Creates necessary theme files for React applications
- Handles theme overrides and references

4. Examples:

- `app-theme.seed`: Shows basic theme usage with the light theme and overrides
- `custom-theme.seed`: Demonstrates more complex theme definitions including:
 - Creating custom themes from scratch
 - Extending existing themes
 - Detailed theme property configurations

5. React Integration:

- Provides a React context-based theme provider
- Includes TypeScript support
- Generates CSS variables for style application
- Offers a `useTheme` hook for consuming theme values

6. Testing: The system includes comprehensive tests for:

- Theme compilation
- CSS generation
- React context generation
- Theme parsing and extraction
- Override application
- Default theme values

To get a complete picture, it might be helpful to also see: - The contents of `src/stdlib/themes.seed` and the base theme files - Any React component examples using the theme system - Documentation about the Seed specification format itself

Would you like me to elaborate on any of these aspects or should we look at additional files?

Tokens: 4.8k sent, 387 received. Cost: \$0.02 message, \$0.03 session.

Add `src/stdlib/themes.seed` to the chat? (Y)es/(N)o/(D)on't ask again [Yes]: y

From analyzing the files, this repository appears to have a comprehensive theming system with several key components:

1. Theme Definition System:

- Uses a custom `".seed"` file format to define themes
- Supports a hierarchical theme system with inheritance
- Has built-in themes (base, light, dark) in `src/stdlib/themes/`
- Allows custom theme definitions with full customization options

2. Theme Properties: The theming system covers:

- Colors (primary, secondary, background, etc.)
- Typography (font families, sizes, weights, line heights)
- Spacing scales
- Borders and shadows
- Smart defaults with the ability to override

3. Tools and Compilation:

- `tools/compile_react_theme.py`: A Python tool that:
 - Converts `.seed` theme definitions to CSS variables
 - Generates React context for theme consumption
 - Creates TypeScript types for theme values
 - Supports theme overrides and extensions

4. Examples:

- `examples/app-theme.seed`: Shows basic theme usage with the light theme and overrides
- `examples/custom-theme.seed`: Demonstrates creating custom themes and extending existing ones
- React theme example showing integration with React applications

5. Testing: Comprehensive test coverage including:

- Theme parsing tests
- CSS generation tests
- React context generation tests
- Theme override and extraction tests

6. React Integration:

- Provides a `ThemeContext.tsx` for React apps
- Includes TypeScript support
- Offers hooks for theme consumption (`useTheme`)
- CSS variable-based implementation

To get a complete picture, it would be helpful to see: - The actual contents of `src/stdlib/themes/base.seed`, `light.seed`, and `dark.seed` - The full implementation of `src/parser.py` referenced in the tests - Any additional example applications showing theme usage

The system appears to be well-designed for both simplicity and flexibility, allowing both quick starts with default themes and deep customization when needed.

Tokens: 4.8k sent, 394 received. Cost: \$0.02 message, \$0.05 session.

Add `src/parser.py` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add `src/stdlib/themes/base.seed` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add `src/stdlib/themes/base.seed` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add `src/stdlib/themes/dark.seed` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add `src/stdlib/themes/dark.seed` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

Add `src/stdlib/themes/light.seed` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: all

Add `src/stdlib/themes/light.seed` to the chat? (Y)es/(N)o/(A)ll/(S)kip all/(D)on't ask again [Yes]: a

This repository implements a comprehensive theming system with the following key components:

1. Theme Definition System:

- Uses `.seed` files to define themes in a hierarchical structure
- Base theme (`themes/base.seed`) defines default values for all theme properties
- Light (`themes/light.seed`) and Dark (`themes/dark.seed`) themes extend the base theme
- Themes include properties for colors, typography, spacing, borders, and shadows

2. Theme Compilation Tools:

- `tools/compile_react_theme.py` - A Python tool that:
 - Converts `.seed` theme definitions into React context and CSS variables
 - Generates `ThemeContext.tsx` for React theme provider
 - Creates CSS variables in `theme.css`
 - Handles theme inheritance and overrides

3. Examples:

- `examples/app-theme.seed` - Simple example using the light theme with overrides
- `examples/custom-theme.seed` - More complex example showing:
 - Custom theme definition from scratch

- Theme extension (modernLight extends light)
- Theme usage in an app with overrides

4. React Integration:

- Generated ThemeContext.tsx provides:
 - Theme context provider
 - useTheme hook for accessing theme values
 - TypeScript types for theme values
- CSS variables for styling components

5. Testing:

- Comprehensive test suite covering:
 - Theme compilation (test_react_theme_compiler.py)
 - Theme specification parsing (test_theme_spec.py)
 - CSS generation
 - React context generation
 - Theme overrides and inheritance

The system allows for: - Using predefined themes (light/dark) - Creating custom themes - Extending existing themes - Applying theme overrides at the app level - Type-safe theme usage in React applications

Would you like me to elaborate on any specific aspect of the system?

Tokens: 6.8k sent, 372 received. Cost: \$0.03 message, \$0.08 session.

show me an example of a theme and how to compile and integrate it into a react app

I'll show you a complete example from theme definition to React integration.

1. First, let's create a custom theme file my-theme.seed:

```
app MyApp {
  ui {
    // Use light theme as base
    theme: "light"

    // Custom overrides
    overrides: {
      colors.primary: "#0070f3"
      colors.secondary: "#7928ca"
      typography.fontSize.base: "16px"
      typography.fontFamily.base: "Inter, system-ui, sans-serif"
      shadows.lg: "0 20px 25px -5px rgba(0, 0, 0, 0.1)"
    }
  }
}
```

2. Compile the theme using the provided tool:

```
python tools/compile_react_theme.py my-theme.seed --output src/theme
```

This will generate two files:

theme.css:

```
:root {
  --colors-primary: #0070f3;
  --colors-secondary: #7928ca;
  --colors-background: #ffffff;
```

```

--colors-surface: #f8fafc;
--colors-text: #1e293b;
/* ... other variables ... */
--typography-fontSize-base: 16px;
--typography-fontFamily-base: Inter, system-ui, sans-serif;
--shadows-lg: 0 20px 25px -5px rgba(0, 0, 0, 0.1);
}

```

ThemeContext.tsx:

```

import { createContext, useContext, ReactNode } from 'react';

export const theme = {
  colors: {
    primary: "#0070f3",
    secondary: "#7928ca",
    background: "#ffffff",
    // ... other colors
  },
  typography: {
    fontSize: {
      base: "16px",
      // ... other sizes
    },
    fontFamily: {
      base: "Inter, system-ui, sans-serif",
      // ... other families
    }
  },
  // ... other theme properties
} as const;

export type Theme = typeof theme;

const ThemeContext = createContext<Theme>(theme);

export function ThemeProvider({ children }: { children: ReactNode }) {
  return (
    <ThemeContext.Provider value={theme}>
      {children}
    </ThemeContext.Provider>
  );
}

export function useTheme() {
  return useContext(ThemeContext);
}

```

3. Use in your React app:

```

// App.tsx
import { ThemeProvider } from './theme/ThemeContext';
import './theme/theme.css';

function App() {

```

```

return (
  <ThemeProvider>
    <MyComponents />
  </ThemeProvider>
);
}

4. Use the theme in components:

// Using CSS variables
const Button = styled.button`
  background-color: var(--colors-primary);
  color: var(--colors-background);
  font-family: var(--typography-fontFamily-base);
  font-size: var(--typography-fontSize-base);
  padding: var(--spacing-md);
  border-radius: var(--borders-radius-md);
  box-shadow: var(--shadows-lg);
`;

// Using React hook
function ThemedComponent() {
  const theme = useTheme();

  return (
    <div style={{
      color: theme.colors.primary,
      fontFamily: theme.typography.fontFamily.base
    }}>
      Themed content
    </div>
  );
}

```

The benefits of this approach:

1. Type-safe theme values in TypeScript
2. CSS variables for better performance
3. Consistent theme values across components
4. Easy theme switching and customization
5. Inheritance from base themes
6. Central theme management

You can also create multiple themes and switch between them, or create responsive themes that change based on media queries or user preferences.

Tokens: 7.2k sent, 802 received. Cost: \$0.03 message, \$0.11 session.

Add examples/react-theme/App.tsx to the chat? (Y)es/(N)o/(D)on't ask again [Yes]: y

litellm.InternalServerError: AnthropicException - Overloaded. Handle with litellm.InternalServerError.

The API provider's servers are down or overloaded.

Retrying in 0.2 seconds...

I'll walk you through the complete process using an example:

1. First, define your theme in a .seed file (e.g., app-theme.seed):

```

app MyApp {
  ui {

```

```

// Use the light theme as base
theme: "light"

// Override some colors and typography
overrides: {
  colors.primary: "#0070f3"
  colors.secondary: "#7928ca"
  typography.fontSize.base: "16px"
  typography.fontFamily.base: "Inter, system-ui, sans-serif"
}
}
}

```

2. Compile the theme using the provided tool:

```
python tools/compile_react_theme.py examples/app-theme.seed --output examples/react-theme
```

This generates two files: - theme.css with CSS variables - ThemeContext.tsx with React context and types

3. In your React app, use the theme like in the example App.tsx:

```
import { ThemeProvider, useTheme } from "../ThemeContext";
import "../theme.css";
```

```

// Example component using the theme
function DemoCard({ title, children }) {
  const theme = useTheme();
  return (
    <div style={{
      backgroundColor: theme.colors.surface,
      padding: theme.spacing.lg,
      borderRadius: theme.borders.radius.lg,
      boxShadow: theme.shadows.md,
    }}>
      <h2 style={{
        color: theme.colors.text,
        fontSize: theme.typography.fontSize.xl,
      }}>{title}</h2>
      {children}
    </div>
  );
}

```

```

// Wrap your app with ThemeProvider
function App() {
  return (
    <ThemeProvider>
      <DemoCard title="Theme Demo">
        <Button primary>Primary Button</Button>
        <Button>Secondary Button</Button>
      </DemoCard>
    </ThemeProvider>
  );
}

```

The example App.tsx shows several ways to use the theme:

1. Direct style properties:

```
style={{
  backgroundColor: theme.colors.background,
  padding: theme.spacing.xl,
  fontFamily: theme.typography.fontFamily.base
}}
```

2. Compound components like DemoCard and Button that use theme values

3. Responsive design using theme values:

```
style={{
  display: "flex",
  gap: theme.spacing.md,
  flexWrap: "wrap"
}}
```

Key benefits: - Type-safe theme access via useTheme hook - Consistent styling across components - Easy theme switching (light/dark) - CSS variables for performance - Customizable via theme overrides

The system supports both direct style usage and CSS variables, allowing you to choose the most appropriate approach for each use case.

Would you like me to explain any specific aspect in more detail?

Tokens: 8.0k sent, 580 received. Cost: \$0.03 message, \$0.14 session.

can this be changed to make the theming system more like tailwind css?

Yes, the theming system could be modified to be more Tailwind-like in several ways:

1. Modify the base theme (src/stdlib/themes/base.seed) to match Tailwind's design system:

```
theme default {
  colors {
    // Tailwind-style color scales
    gray-50: "#f9fafb"
    gray-100: "#f3f4f6"
    gray-200: "#e5e7eb"
    // ... etc

    blue-500: "#3b82f6"
    blue-600: "#2563eb"
    // ... etc
  }

  spacing {
    // Tailwind-style spacing scale
    0: "0"
    0.5: "0.125rem"
    1: "0.25rem"
    1.5: "0.375rem"
    2: "0.5rem"
    // ... etc
  }

  fontSize {
    // Tailwind-style sizes
```

```

xs: ["0.75rem", "1rem"]
sm: ["0.875rem", "1.25rem"]
base: ["1rem", "1.5rem"]
// ... etc
}
}

```

2. Modify the theme compiler (tools/compile_react_theme.py) to generate utility classes:

```

def generate_utility_classes(theme_spec: dict) -> str:
    """Generate Tailwind-style utility classes"""
    css = []

    # Generate color utilities
    for color_name, value in theme_spec['colors'].items():
        css.append(f".bg-{color_name} {{ background-color: {value}; }}")
        css.append(f".text-{color_name} {{ color: {value}; }}")

    # Generate spacing utilities
    for space_name, value in theme_spec['spacing'].items():
        css.append(f".p-{space_name} {{ padding: {value}; }}")
        css.append(f".m-{space_name} {{ margin: {value}; }}")
        css.append(f".gap-{space_name} {{ gap: {value}; }}")

    # Generate typography utilities
    for size_name, [size, line_height] in theme_spec['fontSize'].items():
        css.append(f".text-{size_name} {{ font-size: {size}; line-height: {line_height}; }}")

    return "\n".join(css)

```

3. Update the React components to use utility classes instead of inline styles:

```

// Before:
function DemoCard({ title, children }) {
  const theme = useTheme();
  return (
    <div style={{
      backgroundColor: theme.colors.surface,
      padding: theme.spacing.lg,
      borderRadius: theme.borders.radius.lg,
    }}>
      {children}
    </div>
  );
}

// After:
function DemoCard({ title, children }) {
  return (
    <div className="bg-surface p-4 rounded-lg shadow-md">
      {children}
    </div>
  );
}

```

4. Add a class composition system:

```
// src/theme/classNames.ts
export function cn(...classes: string[]) {
  return classes.filter(Boolean).join(' ');
}

// Usage
function Button({ primary, className, children }) {
  return (
    <button
      className={cn(
        "px-4 py-2 rounded-md",
        primary ? "bg-blue-500 text-white" : "bg-gray-100 text-gray-900",
        className
      )}
    >
      {children}
    </button>
  );
}
```

5. Add responsive and state variants:

```
@media (min-width: 640px) {
  .sm\:text-lg { font-size: 1.125rem; }
}

.hover\:bg-blue-600:hover {
  background-color: #2563eb;
}
```

6. Update the parser to support Tailwind-style configuration:

```
theme config {
  screens {
    sm: "640px"
    md: "768px"
    lg: "1024px"
  }

  variants {
    extend {
      backgroundColor: ["active", "disabled"]
      opacity: ["hover", "focus"]
    }
  }
}
```

Example usage with the new system:

```
function App() {
  return (
    <div className="min-h-screen bg-gray-50 p-6">
      <div className="max-w-md mx-auto space-y-4">
        <Card className="bg-white p-6 rounded-lg shadow-md">
          <h2 className="text-xl font-semibold text-gray-900 mb-4">
            Welcome
          </h2>
        </Card>
      </div>
    </div>
  );
}
```



```

        </h2>
        <Button className="w-full hover:bg-blue-600 transition-colors">
            Get Started
        </Button>
    </Card>
</div>
</div>
);
}

```

This would require: 1. Updating the theme specification format 2. Modifying the compiler to generate utility classes 3. Adding a build step to generate the CSS 4. Creating helper functions for class composition 5. Adding support for responsive and state variants 6. Updating documentation and examples

Would you like me to elaborate on any of these aspects?

Tokens: 8.6k sent, 1.1k received. Cost: \$0.04 message, \$0.18 session.

i would like to change it so that the seed spec is semantic with a fixed schema for most if not all business applications (web based) and it should compile to something that can provide classNames for all the elements in an app.

I understand - you want to create a semantic schema that maps business application UI elements to consistent classNames. Here's how we could modify the system:

1. First, let's define a semantic schema in the base theme:

```

// src/stdlib/themes/base.seed
theme default {
  // Semantic layout components
  layout {
    page {
      container: true
      header: true
      main: true
      sidebar: true
      footer: true
    }

    section {
      container: true
      header: true
      content: true
      footer: true
    }

    grid {
      container: true
      item: true
    }
  }

  // Semantic components for business apps
  components {
    // Data display
    table {
      container: true
    }
  }
}

```

```

    header: true
    row: true
    cell: true
    pagination: true
    sortable: true
    filterable: true
}

dataGrid {
    container: true
    header: true
    toolbar: true
    content: true
    footer: true
}

card {
    container: true
    header: true
    content: true
    footer: true
    actionArea: true
}

// Forms
form {
    container: true
    section: true
    field: true
    label: true
    input: true
    error: true
    help: true
}

// Navigation
nav {
    container: true
    item: true
    active: true
    group: true
}

// Actions
button {
    base: true
    primary: true
    secondary: true
    danger: true
    success: true
    disabled: true
}

```

```

// Feedback
alert {
  info: true
  success: true
  warning: true
  error: true
}

// Status indicators
badge {
  default: true
  success: true
  warning: true
  error: true
}

// Modals and dialogs
dialog {
  overlay: true
  container: true
  header: true
  content: true
  footer: true
}
}
}

```

2. Modify the theme compiler to generate semantic classNames:

```

def generate_semantic_classes(theme_spec: dict) -> str:
    """Generate semantic utility classes"""
    css = []

    def process_component(prefix: str, config: dict):
        for key, value in config.items():
            if isinstance(value, dict):
                process_component(f"{prefix}-{key}", value)
            elif value is True:
                class_name = f".{prefix}-{key}"
                css.append(f"""
                    {class_name} {{
                        /* Base styles */
                        @apply {get_base_styles(prefix, key)};
                        /* Theme-specific styles */
                        {get_theme_styles(prefix, key)}
                    }}
                """)

    # Process layout components
    process_component("layout", theme_spec["layout"])

    # Process semantic components
    process_component("component", theme_spec["components"])

```

```

return "\n".join(css)

def get_base_styles(prefix: str, key: str) -> str:
    """Get base styles for a semantic component"""
    # Define standard styles for common patterns
    patterns = {
        "layout-page-container": "min-h-screen w-full",
        "layout-section-container": "w-full py-6",
        "component-table-container": "w-full overflow-x-auto",
        "component-form-container": "space-y-4",
        # ... etc
    }
    return patterns.get(f"{prefix}-{key}", "")

def get_theme_styles(prefix: str, key: str) -> str:
    """Get theme-specific styles for a semantic component"""
    # These would come from the active theme
    return ""

```

3. Example usage in React components:

```

// Components/DataTable.tsx
function DataTable({ data, columns }) {
    return (
        <div className="component-table-container">
            <div className="component-table-header">
                <div className="component-table-toolbar">
                    <SearchField />
                    <FilterDropdown />
                </div>
            </div>
            <table className="component-table">
                <thead className="component-table-header">
                    { /* ... */ }
                </thead>
                <tbody>
                    {data.map(row => (
                        <tr className="component-table-row">
                            { /* ... */ }
                        </tr>
                    ))}
                </tbody>
            </table>
            <div className="component-table-pagination">
                { /* ... */ }
            </div>
        </div>
    );
}

```

```

// Components/Form.tsx
function BusinessForm({ onSubmit }) {
    return (
        <form className="component-form-container">

```

```

    <div className="component-form-section">
      <div className="component-form-field">
        <label className="component-form-label">
          Name
        </label>
        <input className="component-form-input" />
        <span className="component-form-error">
          {/* error message */}
        </span>
      </div>
    </div>

    <div className="component-form-footer">
      <button className="component-button-secondary">
        Cancel
      </button>
      <button className="component-button-primary">
        Submit
      </button>
    </div>
  </form>
);
}

// Pages/Dashboard.tsx
function DashboardPage() {
  return (
    <div className="layout-page-container">
      <header className="layout-page-header">
        {/* ... */}
      </header>

      <main className="layout-page-main">
        <section className="layout-section-container">
          <div className="layout-grid-container">
            <div className="layout-grid-item">
              <div className="component-card-container">
                {/* ... */}
              </div>
            </div>
          </div>
        </section>
      </main>
    </div>
  );
}

```

4. Add a helper to compose classNames with variants:

```

// theme/classNames.ts
export function getComponentClass(
  component: string,
  element: string,
  variants?: {
    size?: 'sm' | 'md' | 'lg',

```

```

    state?: 'active' | 'disabled' | 'loading',
    intent?: 'primary' | 'secondary' | 'danger'
  }
) {
  const base = `component-${component}-${element}`;
  const variantClasses = variants ?
    Object.entries(variants)
      .map(([key, value]) => `${base}-${key}-${value}`)
      .join(' ')
      : '';

  return `${base} ${variantClasses}`.trim();
}

// Usage
<button className={getComponentClass('button', 'base', {
  size: 'lg',
  intent: 'primary',
  state: 'loading'
})}>
  Submit
</button>

```

This approach provides: - Semantic, predictable classNames - Consistent styling across the application - Type-safe component variants - Easy theme switching - Standard layouts and components for business apps

Would you like me to elaborate on any part of this approach?

Tokens: 9.8k sent, 1.5k received. Cost: \$0.05 message, \$0.24 session.

the idea is to be able to generate theme separate from the rest of the application. if the theme is updated, the rest of the app shouldn't have to be updated. and vice versa.

Ah, I understand better now. We want to decouple the theme generation from the application code, where:

1. The theme system defines semantic structures (like "data-table", "form-field", etc.)
2. Theme changes only require updating theme files and regenerating CSS
3. Application code remains stable using semantic classNames

Here's how we could restructure this:

```

# themes/business-app.seed
theme business-app {
  // Semantic definitions that map to any business app
  components {
    data-table {
      wrapper: {
        base: true
        bordered: true
        compact: true
      }
      header: {
        cell: true
        sortable: true
      }
      body: {
        row: true

```

```

        cell: true
        striped: true
        hoverable: true
    }
    footer: {
        pagination: true
    }
}

form {
    group: {
        base: true
        inline: true
        stacked: true
    }
    label: {
        base: true
        required: true
    }
    input: {
        base: true
        invalid: true
        disabled: true
    }
}

// ... other semantic components
}
}

```

The theme compiler would generate two key outputs:

1. CSS Variables and Classes:

```

/* Generated theme.css */
:root {
    /* Theme variables */
    --data-table-border-color: #e2e8f0;
    --form-input-border: 1px solid #cbd5e1;
    /* ... */
}

/* Semantic classes */
.data-table-wrapper {
    /* Base styles that won't change with theme */
    display: table;
    width: 100%;
}

.data-table-wrapper--bordered {
    border: var(--data-table-border-color);
}

.form-input {
    border: var(--form-input-border);
}

```

```
}
/* ... */
```

2. TypeScript Types:

```
// Generated theme.d.ts
export type SemanticComponent =
  | 'data-table'
  | 'form'
  | 'button'
  // ...

export type ComponentVariant = {
  'data-table': {
    wrapper: ['base', 'bordered', 'compact']
    header: ['cell', 'sortable']
    // ...
  }
  'form': {
    group: ['base', 'inline', 'stacked']
    // ...
  }
  // ...
}
```

Then the application code would use stable semantic classNames:

```
// Application code that doesn't change with theme updates
function DataTable({ data, bordered = true }) {
  return (
    <div className="data-table-wrapper data-table-wrapper--bordered">
      <div className="data-table-header">
        {/* ... */}
      </div>
    </div>
  );
}

function FormField({ label, required }) {
  return (
    <div className="form-group">
      <label className={`form-label ${required ? 'form-label--required' : ''}`}>
        {label}
      </label>
      <input className="form-input" />
    </div>
  );
}
```

The workflow would be:

1. Update theme:

```
# Update theme files
vim themes/business-app.seed
```



```
# Regenerate theme assets
```

```
seed-theme compile themes/business-app.seed --output src/theme
```

2. The generated files update, but application code stays the same because it uses semantic classNames that don't change.

This provides: - Complete separation of theme and application code - Semantic classNames that represent business UI concepts - Type safety for component variants - Ability to switch themes without touching application code - Standard patterns for common business UI elements

Would you like me to elaborate on any part of this approach or show more examples?

Tokens: 11k sent, 772 received. Cost: \$0.05 message, \$0.28 session.

Chapter 5

CONTRIBUTING.md

Chapter 6

Contributing to SeedML

Thank you for your interest in contributing to SeedML! This document provides guidelines and information for contributors.

Getting Started

1. Fork the repository
2. Clone your fork
3. Create a new branch for your work
4. Make your changes
5. Submit a pull request

Development Setup

```
# Install from source
git clone https://github.com/seedml/seedml.git
cd seedml
pip install -e .

# Set your API key
export ANTHROPIC_API_KEY='your-api-key'

# Run tests
python -m pytest

# Build documentation
mkdocs build
```

CLI Development

The SeedML CLI tool generates applications from specifications:

```
# Generate application from spec
seedml myapp.seed

# Show help
seedml --help
```

```
# Show version
seedml --version
```

Options:

```
--api-key KEY      Anthropic API key (or use ANTHROPIC_API_KEY env var)
--output DIR       Output directory (default: ./generated)
--verbose          Show detailed output
```

Contribution Areas

We welcome contributions in these areas:

1. Language Design
 - Syntax improvements
 - New features
 - Pattern recognition
2. Compiler Development
 - Parser improvements
 - Code generation
 - Optimization
3. Documentation
 - Examples
 - Tutorials
 - API documentation
4. Tools
 - IDE plugins
 - Development tools
 - Testing utilities

Code Style

- Follow existing patterns
- Add tests for new features
- Document your changes
- Keep commits focused

Pull Request Process

1. Update documentation
2. Add tests if needed
3. Ensure all tests pass
4. Update CHANGELOG.md
5. Submit PR with clear description

Questions?

- Open an issue for bugs
- Discussions for features
- Discord for community chat

License

By contributing, you agree to license your work under our project's license terms.

Chapter 7

LICENSE-GPL.md

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

[... Full GPL v3 text would go here ...]

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

[... Full terms and conditions would go here ...]

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your Programs

[... Application instructions would go here ...]

Chapter 8

docs/getting-started.md

Chapter 9

Getting Started with SeedML

Installation

```
pip install -e .
```

Quick Start

1. Create a new project:

```
seedml new my-app  
cd my-app
```

2. Edit your app.seed file:

```
app MyApp {  
  entity User {  
    name: string  
    email: email  
    active: bool = true  
  }  
  
  screen Users {  
    list: [name, email, active]  
    actions: [create, edit]  
  }  
}
```

3. Generate and run:

```
seedml generate  
seedml run
```

Next Steps

- Read the [Core Concepts](#)
- Try the [Examples](#)
- Learn about [Patterns](#)

Learning Path

1. Start Here:
 - [Quick Start Guide](#)
 - [First Application](#)
 2. Core Concepts:
 - [Architecture](#)
 - [Type System](#)
 - [Business Rules](#)
 - [UI Patterns](#)
 3. Examples:
 - [Basic CRUD](#)
 - [Business App](#)
 - [Dashboard](#)
 - [SaaS](#)
 4. Reference:
 - [Types](#)
 - [Patterns](#)
 - [CLI](#)
-

Chapter 10

docs/getting-started/basic-concepts.md

Chapter 11

Basic Concepts

The Seed Specification Language (SeedSpec) is built around type-safe, modular concepts that ensure reliability and maintainability.

Core Structure

```
// Import standard library components
import "@stdlib/core"
use { Button, Card } from "@stdlib/components"

app TaskManager {
  // Core domain model with type validation
  entity Task {
    title: string {
      min: 3
      max: 100
    }
    done: boolean
    due: datetime {
      min: now() // Must be in future
    }
  }

  // Type-safe theme definition
  theme MainTheme {
    tokens {
      colors {
        primary: color(#0066cc)
        success: color(green.500)
        error: color(red.500)
      }
      spacing {
        small: size(4px)
        medium: size(8px)
      }
    }
  }
}
```

```

// Strongly typed UI components
component TaskCard {
  required {
    task: Task
    onComplete: function
  }

  styles {
    background: color(white)
    padding: spacing(medium)
    border: {
      width: size(1px)
      style: solid
      color: color(tokens.colors.primary)
    }
  }
}

```

Type System

1. Basic Types

```

types {
  // Text with validation
  string {
    min?: number
    max?: number
    pattern?: regex
  }

  // Numbers with constraints
  number {
    min?: number
    max?: number
    integer?: boolean
  }

  // Dates and times
  datetime {
    min?: datetime
    max?: datetime
    timezone?: string
  }

  // Boolean values
  boolean
}

```

2. UI-Specific Types

```

types {
  // Colors with format validation
  color {

```

```

    type: hex | rgb | hsl | token
    value: string
  }

  // Sizes with units
  size {
    type: px | rem | em
    value: number
    unit: string
  }

  // Typography settings
  font {
    family: string
    size: size
    weight: number
    lineHeight?: number
  }
}

```

Modules and Components

1. Module Structure

```

// Explicit imports
import "@stdlib/core"
use { Button, Input } from "@stdlib/components"

// Explicit exports
export {
  TaskCard,
  TaskList
} from "./components"

// Module definition
module Tasks {
  // Components
  component TaskCard {
    // Type-safe props
    required {
      task: Task
      onComplete: function
    }
  }

  // Styled variants
  variants {
    default: {
      background: color(white)
      padding: spacing(medium)
    }
    highlighted: {
      background: color(yellow.100)
      padding: spacing(large)
    }
  }
}

```

```

    }
  }
}

```

2. Component Schema

```

// Define reusable component contract
schema Button {
  required {
    text: string
    onClick: function
  }

  optional {
    disabled: boolean
    variant: enum {
      values: ["primary", "secondary"]
    }
  }
}

// Implement component
component Button {
  // Implement required schema
  implements: Button

  // Style variants
  variants {
    primary: {
      background: color(tokens.colors.primary)
      text: color(white)
    }
    secondary: {
      background: color(transparent)
      text: color(tokens.colors.primary)
      border: {
        color: color(tokens.colors.primary)
      }
    }
  }
}

```

Best Practices

1. **Use Strong Typing**
 - Always specify types explicitly
 - Add validation constraints
 - Use schema definitions
2. **Structure Code Clearly**
 - One component per file
 - Group related functionality
 - Use clear module boundaries
3. **Follow Type Patterns**
 - Use consistent type definitions

- Validate at compile time
 - Handle all edge cases
4. **Write Clear Contracts**
 - Define explicit schemas
 - Document requirements
 - Use descriptive names
 5. **Think in Components**
 - Break down into small pieces
 - Make components reusable
 - Define clear interfaces
-

Chapter 12

docs/getting-started/first-app.md

Chapter 13

Creating Your First App

This guide walks you through creating your first Seed Spec application.

1. Basic Structure

Start with a simple task management app:

```
app TaskManager {  
  // Core domain model  
  entity Task {  
    title: string  
    done: bool  
    due?: date  
  }  
  
  // User interface  
  screen TaskList {  
    list: [title, done, due]  
    actions: [create, complete]  
  }  
}
```

2. Add Features

Enhance the app with more features:

```
app TaskManager {  
  // Expanded task model  
  entity Task {  
    title: string {  
      min: 3  
      max: 100  
    }  
    description: text  
    done: bool  
    due?: date  
    priority: low/medium/high  
    tags: [string]  
  }  
}
```

```

    // Relations
    assignee?: User
    project?: Project
  }

  // Task list screen
  screen TaskList {
    // Data display
    list: [
      title,
      done,
      due,
      priority,
      assignee
    ]

    // Available actions
    actions: [
      create,
      edit,
      complete,
      delete
    ]

    // List behavior
    filter: [done, priority]
    sort: [due, priority]
    search: [title, description]
  }

  // Task detail screen
  screen TaskDetail {
    // Form fields
    fields: [
      title,
      description,
      due,
      priority,
      assignee
    ]

    // Related data
    related: [comments, history]

    // Available actions
    actions: [save, complete, delete]
  }
}

```

3. Add Business Rules

Define validation and behavior:

```

app TaskManager {
  // Previous code...

  rules {
    // Validation rules
    validate {
      title: required
      due: future
      assignee: exists
    }

    // Business logic
    on_complete {
      // Update project progress
      update: project.progress

      // Notify relevant users
      notify: [assignee, project.owner]

      // Archive if needed
      if: project.autoArchive {
        archive: task
      }
    }

    // Automation rules
    automate {
      // Auto-assign based on tags
      assign_by_tags {
        when: tags.changed
        then: auto_assign
      }

      // Due date reminders
      remind_due {
        when: due - 1.day
        then: notify@assignee
      }
    }
  }
}

```

Next Steps

1. **Add More Features**
 - User authentication
 - Comments system
 - File attachments
2. **Enhance the UI**
 - Custom layouts
 - Rich interactions
 - Mobile views
3. **Integrate Services**

- Email notifications
 - Calendar sync
 - External APIs
-

Chapter 14

docs/getting-started/installation.md

Chapter 15

Installing Seed Spec

Prerequisites

Before installing Seed Spec, ensure you have: - Python 3.8 or higher - pip package manager - Anthropic API key (for Claude access) - Node.js 16+ (for frontend implementations) - Docker (optional, for containerized builds)

Installation

```
# Install from source
git clone https://github.com/fahd-noodleseed/seed-spec.git
cd seed-spec
pip install -e .
```

Configuration

Set your Anthropic API key:

```
export ANTHROPIC_API_KEY='your-api-key'
```

Verify Installation

```
# Should show available commands
seedspec --help

# Should show version number
seedspec --version
```

Current Status

Seed Spec is in early development (v0.1.0). Current features: - Basic application generation from .seed files - Cross-compilation to multiple implementations: - React + FastAPI stack - Vue + Express stack - Angular + NestJS stack - Simple CRUD operations - Template customization support

Many planned features are still in development.

Next Steps

- Follow the [Quick Start Guide](#)
 - Try the [First Application Tutorial](#)
 - Read about [Core Concepts](#)
-

Chapter 16

docs/getting-started/introduction.md

Chapter 17

Introduction to SeedSpec

Why SeedSpec?

Modern software development faces several key challenges: - Business requirements get lost in translation across multiple technical layers - Changes require updates across numerous disconnected components - Different teams (business, frontend, backend) speak different languages - Development requires constant context switching between technologies - Type safety and validation often come too late in the development process

SeedSpec solves these challenges by providing: - A single source of truth for entire applications - Strong type system with compile-time validation - Clear module boundaries and explicit contracts - Natural language-like syntax that maps directly to implementation - AI-native design that works seamlessly with LLMs - Technology-independent specifications that can target any modern stack

Key Features

1. Type Safety First

```
// Types are explicit and validated at compile time
entity User {
  name: string {
    min: 2
    max: 50
  }
  email: string {
    format: email
    unique: true
  }
}
```

2. Clear Module System

```
// Explicit imports and exports
import "@stdlib/core"
use { Button, Card } from "@stdlib/components"

export {
  UserProfile,
  UserSettings
} from "./components"
```

3. Component Schemas

```
// Define reusable contracts
schema Button {
  required {
    text: string
    onClick: function
  }
  optional {
    disabled: boolean
    variant: enum {
      values: ["primary", "secondary"]
    }
  }
}
```

4. Design System Integration

```
// Type-safe themes and styling
theme MainTheme {
  tokens {
    colors {
      primary: color(#0066cc)
      secondary: color(blue.500)
    }
    spacing {
      small: size(4px)
      medium: size(8px)
    }
  }
}
```

Benefits

1. **Early Error Detection**
 - Type errors caught at compile time
 - Clear validation messages
 - No runtime type surprises
2. **Improved Maintainability**
 - Clear module boundaries
 - Explicit dependencies
 - Self-documenting code
3. **Better Collaboration**
 - Shared language between teams
 - Clear contracts and interfaces
 - Consistent patterns
4. **Faster Development**
 - Less boilerplate
 - Automated validation
 - Code generation
 - IDE support

Getting Started

1. Install SeedSpec

```
npm install -g seedspec
```

2. Create a New Project

```
seedspec init my-app  
cd my-app
```

3. Start Coding

```
// app.seed  
app MyApp {  
  // Your app specification here  
}
```

4. Generate Code

```
seedspec generate
```

Ready to learn more? Check out the [Basic Concepts](#) guide.

Chapter 18

docs/getting-started/quick-start.md

Chapter 19

Quick Start Guide

Get up and running with Seed Spec quickly.

1. Define Your App

Create a new file `app.seed`:

```
// app.seed - Core app definition
app MyApp {
  // Define your core entities
  entity Product {
    name: string
    price: money
    stock: int
  }

  // Define your screens
  screen Products {
    list: [name, price, stock]
    actions: [create, edit, delete]
    filter: stock > 0
  }

  // Define component styles using Tailwind tokens
  theme {
    // Product list styles
    list {
      item: {
        bg: "white"
        border: "gray.200"
        hover: {
          bg: "gray.50"
        }
      }
    }
    header: {
      bg: "gray.100"
      text: "gray.700"
      font: "semibold"
    }
  }
}
```

```
}

// Action button styles
button {
  primary: {
    bg: "blue.500"
    text: "white"
    hover: {
      bg: "blue.600"
    }
  }
  secondary: {
    bg: "gray.100"
    text: "gray.800"
    hover: {
      bg: "gray.200"
    }
  }
}

// Form input styles
input {
  base: {
    bg: "white"
    border: "gray.300"
    text: "gray.900"
    focus: {
      border: "blue.500"
      ring: "blue.500"
    }
  }
}
```

2. Run the Generator

```
seed generate app.seed
```

This will: - Generate a full-stack application - Compile theme tokens to Tailwind classes - Set up database and API endpoints - Create React components with theme integration

3. Start the App

seed start

Your app is now running at <http://localhost:3000> with:

- Styled components using your theme
- Dark mode support out of the box
- Responsive design
- Form validation
- API integration

Next Steps

- ## 1. Add more entities and screens

2. Define business rules
3. Customize component themes
4. Add integrations

See the [Full Guide](#) for a more detailed walkthrough.

Chapter 20

docs/core-concepts/architecture.md

Chapter 21

Architecture & Design

The Seed Specification Language transforms specifications into working applications through a carefully designed pipeline architecture.

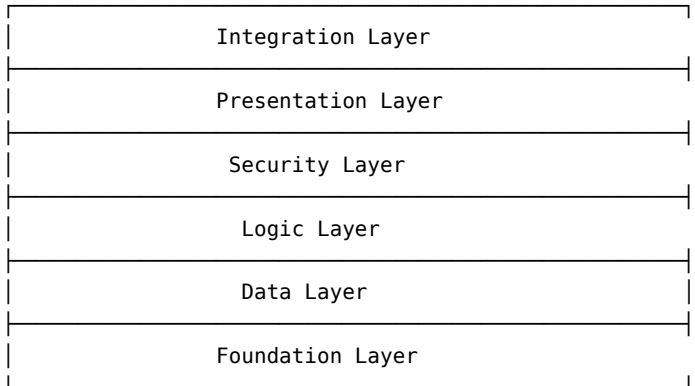
System Architecture

The Seed Specification Language follows a strict layered architecture pattern where each layer has clear responsibilities and boundaries:

Layer Dependencies

<div>Integration Layer</div> <ul style="list-style-type: none">• External services, APIs, webhooks• Location services, mapping providers• Error handling, rate limiting, security
<div>Presentation Layer</div> <ul style="list-style-type: none">• UI components, layouts, navigation• User interactions, forms, validation
<div>Security Layer</div> <ul style="list-style-type: none">• Roles, permissions, access control• Authentication, authorization, audit
<div>Logic Layer</div> <ul style="list-style-type: none">• Business rules, workflows• Computations, validations
<div>Data Layer</div> <ul style="list-style-type: none">• Entities, relationships• CRUD operations, queries
<div>Foundation Layer</div> <ul style="list-style-type: none">• Types, validation rules• Common patterns, base entities

The Seed Specification Language enforces these architectural boundaries:



Each layer builds upon lower layers:

1. **Foundation Layer**
 - Type system
 - Validation rules
 - Computed fields
2. **Data Layer**
 - Independent entities
 - Dependent entities
 - Relationships
3. **Logic Layer**
 - Business rules
 - Workflows
 - Computations
4. **Security Layer**
 - Permissions
 - Roles
 - Access control
5. **Presentation Layer**
 - Screens
 - Components
 - Layouts
6. **Integration Layer**
 - External services
 - APIs
 - Events

Core Components

1. Parser & Validator

The validation pipeline ensures specifications are correct before generation:

```
class Validator:
    def validate_syntax(self, spec):
        """Check YAML syntax and structure"""
        # Verify basic YAML format
        # Check required sections
        # Validate section structure

    def validate_semantics(self, spec):
```

```

        """Verify semantic correctness"""
        # Check entity relationships
        # Validate type usage
        # Verify rule logic

    def resolve_references(self, spec):
        """Resolve all references"""
        # Link entity relationships
        # Resolve type references
        # Connect rule dependencies

```

2. Generation Engine

The generation engine handles interaction with Claude:

```

class Generator:
    def prepare_prompt(self, spec):
        """Transform spec into prompt"""
        # Break into logical chunks
        # Add context and instructions
        # Format for Claude

    def generate_code(self, prompt):
        """Generate via Claude API"""
        # Send prompt to Claude
        # Handle response streaming
        # Manage rate limits
        # Retry on failures

    def process_response(self, response):
        """Process Claude's response"""
        # Parse response format
        # Extract file content
        # Validate output

```

3. Output Processor

The output processor handles file system operations:

```

class OutputProcessor:
    def create_structure(self):
        """Create directory structure"""
        # Make directories
        # Set up config files
        # Initialize git repo

    def write_files(self, files):
        """Write files atomically"""
        # Write to temp location
        # Validate content
        # Move to final location

    def setup_project(self):
        """Configure project"""

```

```
# Install dependencies
# Set up tooling
# Initialize services
```

4. Location Services

```
class LocationServices:
    def geocode(self, address):
        """Convert address to coordinates"""
        # Validate address format
        # Call geocoding service
        # Cache results

    def reverse_geocode(self, lat, lng):
        """Convert coordinates to address"""
        # Validate coordinates
        # Call reverse geocoding
        # Format response

    def validate_region(self, points):
        """Validate geographic region"""
        # Check boundary validity
        # Compute area
        # Verify constraints
```

Design Principles

1. Separation of Concerns

Each component has clear responsibilities: - **Parser**: Understanding specifications - **Generator**: Code generation - **Output**: File system management

2. Reliability

Multiple layers ensure reliable operation: - Validation before generation - Atomic file operations - Error recovery - Output verification

3. Extensibility

The system is designed for extension: - Custom validators - Template overrides - Output adapters - Plugin system

Implementation Details

1. Validation Pipeline

```
def validate_specification(spec):
    # 1. Basic syntax
    validate_yaml_syntax(spec)

    # 2. Schema validation
    validate_against_schema(spec)
```

```
# 3. Semantic checks
validate_semantics(spec)

# 4. Reference resolution
resolve_all_references(spec)
```

2. Generation Process

```
def generate_application(spec):
    # 1. Prepare context
    context = prepare_generation_context(spec)

    # 2. Generate components
    components = []
    for component in spec.components:
        result = generate_component(component, context)
        components.append(result)

    # 3. Post-process
    post_process_components(components)
```

```
# Maps Component Generation
def generate_maps_components(spec):
    # 1. Analyze location usage
    location_fields = find_location_fields(spec)

    # 2. Generate components
    for field in location_fields:
        # Generate picker component
        # Generate view component
        # Generate search component

    # 3. Generate services
    generate_location_services()
    generate_geocoding_pipeline()

    # 4. Setup integration
    configure_maps_provider()
    setup_caching()
```

3. File Management

```
def manage_output(components):
    # 1. Prepare directories
    setup_directory_structure()

    # 2. Write files
    with atomic_writer() as writer:
        for component in components:
            writer.write_component(component)

    # 3. Verify output
    verify_written_files()
```

```
def handle_location_data(location):
    # 1. Validation
    validate_coordinates(location)

    # 2. Geocoding
    with geocoding_client() as client:
        result = client.geocode(location)

    # 3. Data enhancement
    enhance_location_data(result)

    # 4. Storage preparation
    prepare_for_storage(result)
```

Error Handling

The system uses a comprehensive error handling approach:

1. **Validation Errors**
 - Syntax errors
 - Semantic errors
 - Reference errors
2. **Generation Errors**
 - API failures
 - Context limits
 - Invalid output
3. **Output Errors**
 - File system errors
 - Permission issues
 - Verification failures

Cross-Compilation Architecture

The Seed Specification Language uses a sophisticated cross-compilation pipeline to transform declarative specifications into implementation-specific code:

Cross-Compiler Components

<p>Specification Parser</p> <ul style="list-style-type: none"> • YAML/JSON parsing • Schema validation • Intermediate representation
<p>Template Engine</p> <ul style="list-style-type: none"> • Implementation-specific templates • Component generation • Layout processing
<p>Code Generator</p> <ul style="list-style-type: none"> • Target platform adaptation • Project structure generation • Dependency management

The cross-compiler follows these key principles:

1. **Clean Separation**
 - Parser independent of implementation
 - Template system isolation
 - Generator modularity
2. **Implementation Support**
 - Multiple frontend frameworks (React, Vue, Angular)
 - Backend technology options
 - Database flexibility
3. **Template Organization**
 - Component templates
 - Layout patterns
 - Style definitions
 - Configuration templates

Cross-Compilation Process

```
class CrossCompiler:
    def compile(self, spec):
        """Transform spec to implementation"""
        # 1. Parse and validate
        ir = self.parser.parse(spec)

        # 2. Load templates
        templates = self.template_engine.load()

        # 3. Generate code
        self.generator.generate(ir, templates)
```

Future Directions

1. Near-term Improvements

- **Deterministic Generation**
 - Local generation options
 - Template-based generation
 - Hybrid approaches
- **Enhanced Validation**
 - Deep semantic analysis
 - Cross-reference checking
 - Custom rule validation

2. UI Patterns & Components

```
# Standard Screen Types
screen List {
    list: [field1, field2]      # Fields to display
    actions: [create, edit]    # Basic actions
}

screen Form {
```

```

    form: [field1, field2]      # Form fields
    actions: [save, cancel]    # Form actions
  }

  screen Detail {
    content: [field1, field2]  # Content fields
    actions: [edit, delete]    # Item actions
  }

  screen Dashboard {
    summary: [metric1, metric2] # Key metrics
    lists: [recent, popular]    # Data lists
  }

  # Layout Patterns
  screen OrderDetail {
    layout: split
    left: [customer, items]
    right: [summary, actions]
  }

  # Built-in Features
  features: {
    search: true      # Search functionality
    sort: true        # Column sorting
    pagination: true  # Page navigation
  }

```

3. Research Areas

- **Prompt Engineering**
 - Context optimization
 - Output consistency
 - Token efficiency
- **Code Quality**
 - Style consistency
 - Best practice enforcement
 - Security patterns
- **Location Optimization**
 - Smart geocoding caching
 - Efficient region calculations
 - Clustering algorithms
 - Distance matrix optimization

3. Tooling

- **Development Tools**
 - IDE integration
 - Live preview
 - Debug tools
- **Deployment**
 - CI/CD integration
 - Container support
 - Cloud deployment

Contributing

The architecture is designed for contribution in these areas:

1. **Core Components**
 - Validators
 - Generators
 - Processors
2. **Extensions**
 - Templates
 - Plugins
 - Tools
3. **Documentation**
 - Examples
 - Tutorials
 - Reference # Architecture

SeedML generates scalable architectures through intent-focused patterns that automatically implement proven practices.

Core Concepts

```
app ScalableApp {  
  # Declare architecture needs  
  architecture {  
    style: microservices    # Application pattern  
    scale: auto             # Infrastructure  
    regions: [us, eu]      # Distribution  
    resilience: high        # Reliability  
  }  
  
  # Intent-focused services  
  service Orders {  
    type: core              # Service classification  
    scale: high             # Resource allocation  
    storage: dedicated      # Data patterns  
  }  
}
```

Key Features

1. Application Patterns

```
architecture {  
  # Smart architectural choices  
  style: microservices  
  patterns: [  
    cqrs,                  # Command/Query  
    event_sourcing,        # State management  
    api_gateway            # Access control  
  ]  
}
```

2. Scalability

```
scale {  
  # Automatic scaling  
  compute: auto      # Resources  
  storage: replicated # Data  
  cache: distributed  # Performance  
  
  limits: {  
    cpu: 80%          # Thresholds  
    memory: 70%       # Monitoring  
  }  
}
```

3. Resilience

```
resilience {  
  # Built-in reliability  
  failover: automatic # Recovery  
  backup: continuous  # Data protection  
  monitoring: complete # Observability  
  
  sla: {  
    uptime: 99.9%      # Availability  
    latency: 100ms     # Performance  
  }  
}
```

4. Distribution

```
regions {  
  # Global deployment  
  primary: us-east  
  replicas: [eu, asia]  
  
  routing: {  
    strategy: latency # Smart routing  
    failover: nearest # Reliability  
  }  
}
```

Best Practices

1. **Cloud Native**
 - Container based
 - Auto scaling
 - Service mesh
2. **Reliability First**
 - High availability
 - Disaster recovery
 - Performance monitoring
3. **Future Ready**

- Modular design
 - Easy scaling
 - Tech flexibility
-

Chapter 22

docs/core-concepts/business-rules.md

Chapter 23

Business Rules

The Seed Specification Language provides a simplified approach to business rules focused on common use cases and rapid prototyping.

Core Concepts

1. Basic Validation

```
entity User {
  email: email
  password: string

  rules {
    validate {
      email: required,
      password: length >= 8
    }
  }
}
```

2. Simple State Transitions

```
entity Task {
  status: string = "todo"

  rules {
    start {
      validate: status == "todo"
      then: updateStatus("in-progress")
    }

    complete {
      validate: status == "in-progress"
      then: updateStatus("done")
    }
  }
}
```

3. Basic Computations

```
entity Invoice {
  items: [InvoiceItem]

  rules {
    calculate {
      then: [
        updateSubtotal(sum(items.amount)),
        updateTotal(subtotal + tax)
      ]
    }
  }
}
```

Common Patterns

1. Field Validation

```
entity Product {
  rules {
    validate {
      name: required,
      price: positive,
      stock: minimum(0)
    }
  }
}
```

2. Cross-field Validation

```
entity Booking {
  rules {
    validate {
      endDate: after(startDate),
      capacity: lessThan(maxCapacity)
    }
  }
}
```

3. Simple Workflows

```
entity Leave {
  status: string = "requested"

  rules {
    approve {
      validate: status == "requested"
      then: [
        updateStatus("approved"),
        notifyEmployee
      ]
    }
  }
}
```

```

    }

    reject {
      validate: status == "requested"
      then: [
        updateStatus("rejected"),
        notifyEmployee
      ]
    }
  }
}

```

Best Practices

1. Keep Validations Simple

```

// DO - Use simple validations
entity Order {
  rules {
    submit {
      validate: [
        items.length > 0,
        total > 0
      ]
    }
  }
}

```

2. Use Clear State Transitions

```

// DO - Simple state changes
entity Task {
  rules {
    complete {
      validate: status == "active"
      then: updateStatus("completed")
    }
  }
}

```

3. Minimize Complexity

- Focus on common use cases
- Avoid complex validation chains
- Keep workflows linear
- Use simple state machines

4. Prefer Convention

- Use standard validation patterns
- Follow common state transitions
- Apply consistent naming

- Leverage built-in behaviors

Key Benefits

1. **Rapid Development**
 - Quick to implement
 - Easy to understand
 - Fast to modify
 2. **Reduced Errors**
 - Simple validation
 - Clear state flow
 - Standard patterns
 3. **Better Maintenance**
 - Less complexity
 - Standard approaches
 - Clear intent
-

Chapter 24

docs/core-concepts/core-qualities.md

Chapter 25

Core Qualities

SeedML is built on four fundamental qualities that make it uniquely suited for AI-native development:

1. AI-First Design

```
# Clear patterns that map to natural language
entity Order {
  # AI understands these concepts naturally
  status: enum(draft, submitted, approved)
  total: money
  customer: reference

  # Intent is clear to both AI and humans
  validate: {
    total: positive
    customer: verified
  }
}
```

Key Aspects: - Designed for AI generation and modification - Natural language mapping - Context-preserving structure
- Minimal but unambiguous syntax

2. Intent Over Implementation

```
# Express what you want, not how to build it
screen Orders {
  # High-level patterns imply implementation
  list: [date, customer, total, status]
  actions: [create, approve]
  features: [search, filter, export]
}
```

Key Aspects: - Focus on business goals - Hide technical complexity - Smart pattern recognition - Progressive disclosure

3. Single Source of Truth

```
# One file describes everything
app OrderSystem {
  # All aspects in one place
  data: {
    entities: [Order, Customer]
    storage: cloud
  }

  ui: {
    screens: [dashboard, orders]
    theme: modern
  }

  rules: {
    workflow: [approve, fulfill]
    security: role_based
  }
}
```

Key Aspects: - Complete system specification - No redundancy - Automatic consistency - Clear dependencies

4. Smart Defaults

```
# Best practices built-in
entity User {
  # These imply proper handling
  email: email      # Validation included
  password: secure  # Hashing automatic
  role: admin       # Permissions set

  # Override only when needed
  name: string {
    min: 2,
    max: 50
  }
}
```

Key Aspects: - Production patterns included - Override when needed - Progressive complexity - Secure by default

Why These Matter

1. **Faster Development**
 - AI generates more accurately
 - Less boilerplate
 - Fewer decisions needed
2. **Better Quality**
 - Consistent patterns
 - Built-in best practices
 - Reduced errors
3. **Future Ready**
 - AI-native architecture

- Technology independent
- Easy to evolve # Core Qualities

SeedML ensures essential software qualities through intent-focused patterns that automatically implement best practices.

Core Concepts

```
app QualityApp {
  # Declare quality needs
  qualities {
    performance: high      # Speed focus
    reliability: 99.9%     # Uptime target
    maintainable: true     # Code quality
    testable: complete     # Coverage
  }

  # Quality-focused features
  api Orders {
    cache: smart           # Performance
    retry: automatic       # Reliability
    docs: generated        # Maintainability
    tests: integration     # Quality
  }
}
```

Key Features

1. Performance

```
performance {
  # Automatic optimization
  cache: {
    strategy: smart      # Caching
    invalidate: auto     # Freshness
  }

  optimize: {
    queries: true        # Database
    assets: true         # Frontend
    api: true            # Backend
  }
}
```

2. Reliability

```
reliability {
  # Built-in stability
  errors: {
    handling: complete   # Error management
    recovery: automatic  # Self-healing
    reporting: detailed  # Monitoring
  }
}
```

```

}

testing: {
  unit: required      # Code quality
  integration: auto    # System health
  performance: load    # Capacity
}
}

```

3. Maintainability

```

maintainable {
  # Code quality focus
  structure: {
    modular: true      # Organization
    documented: auto    # Understanding
    consistent: true    # Standards
  }

  practices: {
    clean: true         # Code quality
    tested: true         # Verification
    reviewed: true      # Quality control
  }
}

```

4. Testability

```

testing {
  # Comprehensive testing
  coverage: {
    unit: 80%          # Code tests
    integration: 90%    # System tests
    e2e: critical       # User flows
  }

  automation: {
    ci: full            # Integration
    deployment: safe    # Release
    monitoring: live    # Production
  }
}

```

Best Practices

1. **Quality by Design**
 - Built into patterns
 - Automated checks
 - Continuous verification
2. **Performance First**
 - Smart optimization
 - Efficient patterns

- Regular monitoring
3. **Future Ready**
- Clean architecture
 - Full testing
 - Easy maintenance
-

Chapter 26

docs/core-concepts/integration.md

Chapter 27

Integration

Seed Spec provides powerful integration capabilities to connect with external services and systems.

Core Integration Patterns

1. Smart Authentication

```
app MyApp {  
  // OAuth-based authentication  
  auth {  
    provider: google  
    scopes: [profile, email]  
    roles: [user, admin]  
  }  
}
```

2. Data Integration

```
storage {  
  // File storage  
  files {  
    provider: s3  
    bucket: uploads  
    types: [image, pdf]  
  }  
  
  // Database  
  database {  
    provider: postgres  
    replicas: 2  
    backup: daily  
  }  
}
```

3. Service Communication


```

services {
  // REST APIs
  api {
    stripe {
      type: rest
      base: "https://api.stripe.com/v1"
      auth: bearer
    }

    weather {
      type: rest
      base: "https://api.weather.com"
      auth: apikey
    }
  }

  // Message queues
  queue {
    orders {
      type: sqs
      fifo: true
    }

    notifications {
      type: rabbitmq
      durable: true
    }
  }
}

```

4. Location Services

```

location {
  // Geocoding
  geocoding {
    provider: google
    cache: true
  }

  // Routing
  routing {
    provider: mapbox
    mode: [driving, walking]
  }
}

```

5. Event Handling

```

events {
  // Webhooks
  webhooks {
    stripe {

```

```

    endpoint: "/webhooks/stripe"
    events: [payment.success, refund]
  }
}

// Real-time
realtime {
  provider: pusher
  channels: [orders, chat]
}
}

```

6. Maps Integration

```

maps {
  // Map provider
  provider: mapbox

  // Features
  features {
    search: true
    routing: true
    clustering: true
  }

  // Styling
  style: streets-v11
  controls: [zoom, fullscreen]
}

```

Best Practices

1. Role-Based Access

```

"javascript // Location-based Integration integrate { // Map provider configuration maps { provider: google apiKey:
env.GOOGLE_MAPS_KEY }

// Location services location { // Geocoding service geocoding { provider: google cache: true rateLimit: 100 }

// Distance calculations
distance {
  provider: google
  mode: [driving, walking]
  units: metric
}

// Access control access { // Role-based rules rules { admin: [all] manager: [view, edit] user: [view] }

// Location-based rules
location {
  required: true
  maxDistance: 100
  unit: kilometers
}

```

}
} }

Chapter 28

**docs/core-concepts/language-
structure.md**

Chapter 29

Seed Specification Language Structure

The Seed Specification Language (SeedML) provides a type-safe, modular way to define consistent design systems and applications.

Core Principles

1. Type Safety First

All values must have explicit types to prevent ambiguity and catch errors early:

```
app TodoApp {
  theme MainTheme {
    tokens {
      colors {
        primary: color(#0066cc)    // Explicit color type
        secondary: color(blue.500) // Color token reference
      }

      spacing {
        small: size(4px)           // Explicit size type
        medium: size(8px)
        large: size(16px)
      }
    }
  }
}
```

2. Modular Design

Clear module system with explicit imports and exports:

```
// Import from standard library
import "@stdlib/core"

// Use specific exports
use { colors, spacing } from theme

// Export your own components
export {
  Button,
```

```
    Card
  } from "../components"
```

3. Component-Based Structure

Components are first-class citizens with schema validation:

```
// Define component schema
schema Button {
  required {
    background: color
    text: color
  }
  optional {
    padding: spacing
    border: border
  }
}

// Implement component
component Button {
  variants {
    primary: {
      background: color(blue.500)
      text: color(white)
      padding: spacing(4)
    }

    secondary: {
      background: color(gray.100)
      text: color(gray.900)
      padding: spacing(4)
    }
  }
}
```

Key Components

1. Type System

Built-in types ensure values are valid:

```
types {
  // Core data types
  string: {
    min: number
    max: number
    pattern?: regex
  }

  number: {
    min?: number
    max?: number
    integer?: boolean
  }
}
```

```

// UI specific types
color: hex | token      // #fff or blue.500
size: px | rem | token  // 16px or 1rem or sm
spacing: size | array   // 16px or [16px, 32px]

// Complex types
entity: {
  fields: [Field]
  rules?: [Rule]
}

screen: {
  layout: grid | list
  components: [Component]
}
}

```

2. Design System

Reusable design tokens with explicit types:

```

app MyApp {
  theme MainTheme {
    // Color palette
    tokens {
      colors {
        primary: color(#0066cc)
        secondary: color(#666666)
        success: color(green.500)
        error: color(red.500)
      }

      // Typography scale
      typography {
        body: {
          size: size(16px)
          lineHeight: size(1.5)
          family: font("Inter")
        }
        heading: {
          size: size(24px)
          lineHeight: size(1.2)
          family: font("Inter")
          weight: number(600)
        }
      }
    }
  }
}

// Business entities
entity User {
  name: string {
    min: 2
    max: 50
  }
}

```

```

    }
    email: email {
      unique: true
    }
  }

  // UI screens
  screen UserList {
    layout: grid(3)
    components: [
      UserCard,
      Pagination
    ]
    actions: [create, edit, delete]
  }
}

```

3. Components

Reusable UI components with variants:

```

component Button {
  // Base styles
  base {
    padding: spacing(4)
    border: none
    borderRadius: size(4px)
  }

  // Variants
  variants {
    primary: {
      background: color(tokens.colors.primary)
      text: color(white)
    }
    secondary: {
      background: color(transparent)
      text: color(tokens.colors.primary)
      border: {
        width: size(1px)
        style: solid
        color: color(tokens.colors.primary)
      }
    }
  }
}

// States
states {
  hover: {
    opacity: number(0.9)
  }
  disabled: {
    opacity: number(0.5)
    cursor: not-allowed
  }
}

```



```
}  
}
```

Best Practices

1. **Use Types Consistently**
 - Always specify value types
 - Use token references when possible
 - Validate values at parse time
 2. **Structure Modules Clearly**
 - One component per file
 - Group related tokens
 - Use explicit imports/exports
 3. **Follow Component Patterns**
 - Define clear schemas
 - Use consistent variant names
 - Handle all states
 4. **Maintain Application Structure**
 - Separate concerns (entities, UI, themes)
 - Use consistent naming
 - Follow domain-driven design principles
-

Chapter 30

docs/core-concepts/modular-apps.md

Chapter 31

Modular App Specifications

The Seed language supports splitting app definitions across multiple files using the `extend` keyword. This enables better organization, team collaboration, and reuse of components.

Basic Usage

```
# core.seed - Core app definition
app MyApp {
  entity User {
    name: string
    email: email
  }
}

# ui.seed - UI components
extend MyApp {
  screen Users {
    list: [name, email]
    actions: [create, edit]
  }
}

# rules.seed - Business rules
extend MyApp {
  rules {
    validateEmail: {
      when: email.changed
      validate: email.format
    }
  }
}
```

Key Benefits

1. **Separation of Concerns**
 - Split specifications by feature
 - Organize by team responsibility

- Maintain focus in each file
2. **Team Collaboration**
 - UI team works on screens
 - Backend team handles data model
 - Rules team manages business logic
 3. **Reusability**
 - Share common components
 - Create feature libraries
 - Mix and match modules

File Organization

Recommended file organization:

```
myapp/
├── core.seed      # Core app definition
├── ui/
│   ├── admin.seed  # Admin screens
│   └── public.seed # Public screens
├── rules/
│   ├── auth.seed   # Auth rules
│   └── billing.seed # Billing rules
└── api/
    └── external.seed # External APIs
```

Loading Order

Files are loaded in this order:

1. Core app definition (app keyword)
2. Extensions (extend keyword) in alphabetical order
3. Feature modules
4. Component libraries
5. Integration modules

Validation

The system ensures:

1. **Consistency**
 - No conflicting definitions
 - Valid references
 - Complete dependencies
2. **Uniqueness**
 - No duplicate entities
 - Unique component names
 - Distinct rule names
3. **Dependencies**
 - Core app exists
 - Required modules present
 - Valid references

Best Practices

1. **File Naming**
 - Use descriptive names
 - Group related files
 - Follow consistent patterns
 2. **Module Size**
 - Keep files focused
 - Split large modules
 - Group related features
 3. **Dependencies**
 - Minimize coupling
 - Clear dependencies
 - Explicit imports
-

Chapter 32

docs/core-concepts/overview.md

Chapter 33

Core Concepts

SeedML is built on simple but powerful concepts that work together:

Modular Apps

Split app definitions across multiple files for better organization and team collaboration. See [Modular Apps](#) for details.

Key Ideas

```
// Everything in one place
app TodoList {
  // Data model
  entity Task {
    title: string
    done: bool
  }

  // Business rules
  rules {
    complete {
      validate: !done
      then: done = true
    }
  }

  // User interface
  screen Tasks {
    list: [title, done]
    actions: [create, complete]
  }
}
```

Core Features

1. **Smart Defaults**
 - Production patterns built-in

- Override only when needed
 - Progressive enhancement
2. **Clear Intent**
 - Express what you want
 - Not how to build it
 - Natural language
 3. **Full Stack**
 - One specification
 - Complete application
 - Modern tech stack

Learn More

1. [Type System](#)
 - Basic types
 - Validation
 - Relationships
 2. [Business Rules](#)
 - Simple validation
 - Clear workflows
 - Computed fields
 3. [UI Patterns](#)
 - Standard layouts
 - Common components
 - Best practices
 4. [Theming](#)
 - Hierarchical themes
 - Simple overrides
 - Visual consistency
-

Chapter 34

docs/core-concepts/security.md

Chapter 35

Security

Seed Spec provides comprehensive security features built into the language.

Core Security Features

1. Authentication

```
app SecureApp {  
  // Authentication configuration  
  auth {  
    // Providers  
    providers: [  
      google {  
        clientId: env.GOOGLE_CLIENT_ID  
        scopes: [profile, email]  
      },  
      github {  
        clientId: env.GITHUB_CLIENT_ID  
        scopes: [user]  
      }  
    ]  
  }  
  
  // Session management  
  session {  
    duration: 24h  
    renewal: true  
    singleDevice: false  
  }  
  
  // Two-factor auth  
  twoFactor {  
    required: true  
    methods: [app, sms]  
  }  
}
```

2. Authorization

```
roles {  
  // Role definitions  
  admin {  
    permissions: [all]  
  }  
  
  manager {  
    permissions: [  
      users.view,  
      users.edit,  
      orders.manage  
    ]  
  }  
  
  user {  
    permissions: [  
      profile.edit,  
      orders.create  
    ]  
  }  
}
```

3. Data Protection

```
protect {  
  // Encryption  
  encrypt {  
    fields: [ssn, creditCard]  
    algorithm: aes-256-gcm  
  }  
  
  // Data masking  
  mask {  
    fields: [email, phone]  
    pattern: "***-***-**{last4}"  
  }  
  
  // Access control  
  access {  
    rules: [  
      "user.id = record.userId",  
      "user.role = 'admin'"  
    ]  
  }  
}
```

4. Audit Logging

```
audit {  
  // What to log  
  track {
```

```

    changes: [create, update, delete]
    access: [view, export]
    auth: [login, logout, failed]
  }

  // Log details
  details {
    user: true
    timestamp: true
    location: true
    changes: diff
  }

  // Retention
  retain {
    duration: 1y
    backup: true
  }
}

```

Security Patterns

1. Data-Level Security

```

entity Order {
  // Fields
  id: uuid
  total: money
  status: pending/paid/shipped

  // Security rules
  security {
    view: ["user.id = userId", "user.role = 'admin'"]
    edit: ["user.role in ['admin', 'manager']"]
    delete: ["user.role = 'admin'"]
  }

  // Field-level security
  fields {
    creditCard {
      view: ["user.role = 'admin'"]
      encrypt: true
    }

    notes {
      edit: ["user.id = assignedTo"]
    }
  }
}

```

2. User Management

```
entity User {  
  // Core fields  
  id: uuid  
  email: email  
  password: password  
  role: admin/manager/user  
  
  // Security features  
  security {  
    password {  
      minLength: 12  
      require: [number, special, mixed]  
      expire: 90d  
    }  
  
    logout {  
      attempts: 5  
      duration: 15m  
    }  
  
    mfa {  
      required: true  
      methods: [app, sms]  
    }  
  }  
}
```

3. Location Privacy

```
“javascript entity UserLocation { // Location data location: location timestamp: datetime accuracy: float  
// Privacy rules privacy { // Precision control precision: city  
// Access rules  
access {  
  exact: ["user.role = 'admin'"]  
  approximate: ["user.role = 'manager'"]  
}  
  
// Retention  
retain {  
  duration: 30d  
  anonymize: true  
}  
} }
```

Chapter 36

docs/core-concepts/smart-defaults.md

Chapter 37

Smart Defaults

SeedML reduces boilerplate through intelligent defaults while maintaining flexibility for custom configurations.

Core Principles

1. Convention Over Configuration

SeedML follows established patterns and best practices by default:

```
entity User {  
  name: string      # Implies: required, indexed  
  email: email       # Implies: unique, validated  
  created: timestamp # Implies: auto-set, immutable  
}
```

2. Progressive Complexity

Start simple and add complexity only when needed:

```
# Simple case - uses all defaults  
entity Product {  
  name: string  
  price: money  
}  
  
# Complex case - custom configuration  
entity Product {  
  name: string {  
    min: 3  
    max: 100  
    format: title_case  
  }  
  price: money {  
    min: 0  
    precision: 2  
    currency: USD  
  }  
}
```

3. Contextual Awareness

Defaults change based on context:

```
entity Order {  
  status: draft->submitted->approved  
  # Implies:  
  # - State machine behavior  
  # - Status validation  
  # - Transition hooks  
  # - Audit logging  
  # - UI status indicators  
}
```

Common Default Patterns

1. Type-Based Defaults

Each type comes with sensible defaults:

- string: Required, trimmed, max length
- email: Unique, validated format
- money: Non-negative, precision(2)
- date: Valid range, proper formatting
- phone: Format validation, optional
- location: Validated coordinates, geocoding, map display
- region: Boundary validation, area calculation
- distance: Unit conversion, formatting

```
# Location type implications
```

```
location: {  
  validation: coordinates  
  geocoding: automatic  
  reverse: on_save  
  format: address  
}
```

```
region: {  
  validation: boundary  
  calculation: area  
  contains: points  
}
```

```
distance: {  
  conversion: automatic  
  display: localized  
}
```

2. UI Patterns

Standard UI components with smart defaults:

```
screen Products {  
  list: [name, price] # Implies:  
  # - Pagination  
  # - Sorting
```



```

# - Search
# - Responsive layout
}

# Map components with smart defaults
screen Locations {
  map: [location] # Implies:
  # - Marker clustering
  # - Bounds fitting
  # - Zoom controls
  # - Mobile gestures
  # - Location search
  # - Responsive layout
}

# Progressive map enhancement
map: {
  basic: location      # Single marker
  multiple: [location] # Clustered markers
  interactive: selector # Location picker
  advanced: {          # Full features
    cluster: true
    search: radius
    draw: regions
  }
}

```

3. Business Logic

Common business patterns are built-in:

```

entity Invoice {
  status: draft->submitted->approved
  # Implies:
  # - State transitions
  # - Validation rules
  # - Notifications
  # - Audit trails
}

entity Store {
  location: location
  # Implies:
  # - Distance calculations
  # - Geocoding pipeline
  # - Region validation
  # - Location indexing
  # - Search optimization
}

# Location-aware rules
rules {
  within_region: true # Region containment
  distance_calc: auto # Distance computation
}

```

```
    geo_index: enabled    # Spatial indexing
}
```

4. Security

Security best practices by default:

```
entity Document {
  access: role.manager
  # Implies:
  # - Role-based access control
  # - Permission checking
  # - Audit logging
  # - Data filtering
}

location_data {
  access: restricted
  # Implies:
  # - Coordinate precision control
  # - Address masking
  # - Usage tracking
  # - API key management
}
```

Overriding Defaults

When defaults don't fit, explicit configuration takes precedence:

```
entity CustomProduct {
  # Override string defaults
  name: string {
    required: false
    max: 500
    format: custom_regex("[A-Z].*")
  }

  # Override money defaults
  price: money {
    min: -1000 # Allow negative
    precision: 4 # 4 decimal places
  }

  # Override timestamp defaults
  created: timestamp {
    auto: false
    mutable: true
  }
}
```

Benefits

1. Faster Development

- Less boilerplate code
 - Fewer decisions needed
 - Quick prototyping
2. Consistency
 - Standard patterns
 - Best practices built-in
 - Uniform behavior
 3. Maintainability
 - Clear override points
 - Documented defaults
 - Centralized configuration
 4. Security
 - Secure by default
 - Best practices enforced
 - Explicit overrides needed
-

Chapter 38

`docs/core-concepts/theming.md`

Chapter 39

Component-Based Theming

Seed Spec provides a component-based theming system that uses Tailwind tokens to define styles. Themes are defined in the standard library (`src/stdlib/themes.seed`) and compile to Tailwind utility classes.

Basic Usage

Define component styles using Tailwind tokens:

```
theme default {
  button {
    primary: {
      bg: "blue.500"           // Uses Tailwind color token
      text: "white"           // Uses Tailwind color token
      hover: {
        bg: "blue.600"       // Uses Tailwind color token
      }
      focus: {
        ring: "blue.500"     // Uses Tailwind color token
      }
    }
  }
}

card {
  bg: "white"                 // Uses Tailwind color token
  border: "gray.200"         // Uses Tailwind color token
  shadow: "sm"               // Uses Tailwind shadow token
  hover: {
    shadow: "md"             // Uses Tailwind shadow token
  }
}
```

Use the compiled theme tokens in components:

```
function Button() {
  const classes = useThemeTokens("button.primary")
  // Compiles to: "bg-blue-500 text-white hover:bg-blue-600 focus:ring-blue-500"
  return <button className={classes}>Click Me</button>
}
```

```
function Card() {
  const classes = useThemeTokens("card")
  // Compiles to: "bg-white border-gray-200 shadow-sm hover:shadow-md"
  return <div className={classes}>Content</div>
}
```

Component Tokens

Theme tokens are organized by component and variant:

Button Variants

```
button {
  primary: {
    bg: "blue.500"
    text: "white"
  }
  secondary: {
    bg: "gray.100"
    text: "gray.800"
  }
}
```

Form Elements

```
input {
  base: {
    bg: "white"
    border: "gray.300"
    focus: {
      border: "blue.500"
      ring: "blue.500"
    }
  }
}
```

Typography

```
typography {
  heading: {
    color: "gray.900"
    font: "sans"
    weight: "bold"
  }
  body: {
    color: "gray.600"
    font: "sans"
  }
}
```

Dark Mode

Define dark mode variants using Tailwind tokens:

```
theme dark extends default {  
  button {  
    primary: {  
      bg: "blue.400"  
      text: "white"  
    }  
  }  
  
  card {  
    bg: "gray.800"  
    border: "gray.700"  
  }  
}
```

The tokens compile to dark mode classes:

```
function Card() {  
  const classes = useThemeTokens("card")  
  // Compiles to: "bg-white dark:bg-gray-800 border-gray-200 dark:border-gray-700"  
  return <div className={classes}>Content</div>  
}
```

Theme Composition

Themes support token composition and inheritance:

```
theme default {  
  // Base tokens can be extended  
  button {  
    base: {  
      px: "4"  
      py: "2"  
      rounded: "md"  
    }  
    primary: {  
      extends: "base" // Inherits base tokens  
      bg: "blue.500"  
      text: "white"  
    }  
  }  
}
```

Smart Defaults

The theme system provides: - Component-based organization of design tokens - Automatic compilation to Tailwind classes - Theme inheritance and composition - Dark mode support out of the box

Chapter 40

docs/core-concepts/type-system.md

Chapter 41

Type System

The Seed Specification Language's type system provides strict typing with explicit validation.

Core Types

```
types {
  // Basic Types
  string {
    min?: number      // Minimum length
    max?: number      // Maximum length
    pattern?: regex   // Regex pattern
    format?: email | url | phone // Format validation
  }

  number {
    min?: number      // Minimum value
    max?: number      // Maximum value
    integer?: boolean // Must be integer
    positive?: boolean // Must be positive
  }

  boolean          // true/false

  // Collection Types
  array {
    type: Type      // Array element type
    min?: number    // Minimum length
    max?: number    // Maximum length
    unique?: boolean // Elements must be unique
  }

  map {
    key: Type       // Key type
    value: Type     // Value type
  }

  enum {
    values: [string] // Allowed values
  }
}
```

```
}  
}
```

UI-Specific Types

```
types {  
  // Color Values  
  color {  
    type: hex | rgb | hsl | token  
    value: string  
  }  
  
  // Size Values  
  size {  
    type: px | rem | em | token  
    value: number  
    unit?: string  
  }  
  
  // Spacing Values  
  spacing {  
    type: size | array  
    value: size | [size, size, size, size]  
  }  
  
  // Typography Values  
  font {  
    family: string  
    size: size  
    weight: number | token  
    style?: normal | italic  
    lineHeight?: number | size  
  }  
  
  // Border Values  
  border {  
    width: size  
    style: solid | dashed | dotted  
    color: color  
    radius?: size  
  }  
}
```

Type Usage

1. Explicit Type Declaration

All values must have explicit types:

```
theme MainTheme {  
  tokens {  
    colors {  
      primary: color(#0066cc)  
      secondary: color(blue.500)  
    }  
  }  
}
```

```

    }

    spacing {
      small: size(4px)
      medium: size(8px)
      large: size(16px)
    }
  }
}

```

2. Component Type Validation

Components define required and optional typed properties:

```

schema Button {
  required {
    background: color
    text: color
    padding: spacing
  }

  optional {
    border: border
    hover: {
      background: color
      text: color
    }
  }
}

```

3. Entity Type Validation

Business entities use type validation:

```

entity User {
  name: string {
    min: 2
    max: 50
    pattern: "[A-Za-z ]+"
  }

  email: string {
    format: email
    unique: true
  }

  age: number {
    min: 0
    max: 150
    integer: true
  }

  roles: array {
    type: enum {
      values: ["user", "admin", "moderator"]
    }
  }
}

```

```

    }
  }
}

```

Type Composition

Types can be composed to create complex structures:

```

// Custom type definitions
types {
  UserRole: enum {
    values: ["user", "admin", "moderator"]
  }

  Address: {
    street: string
    city: string
    country: string
    postal: string {
      pattern: "[0-9]{5}"
    }
  }
}

// Using composed types
entity User {
  profile: {
    name: string
    avatar: string {
      format: url
    }
  }
  address: Address
}

settings: map {
  key: string
  value: string | number | boolean
}

permissions: array {
  type: UserRole
  unique: true
}
}

```

Benefits

1. **Type Safety**
 - Compile-time validation
 - Runtime type checking
 - No implicit conversions
2. **Clear Contracts**
 - Self-documenting schemas

- Explicit validation rules
- IDE support

3. **Error Prevention**

- Early error detection
- Clear error messages
- Validation at parse time

4. **Code Generation**

- Type-safe APIs
 - Database schemas
 - UI components
 - Form validation
-

Chapter 42

docs/patterns/advanced-patterns.md

Chapter 43

Advanced SeedML Patterns

This guide covers advanced patterns and techniques for building sophisticated applications with SeedML.

Multi-Tenant Architecture

```
app MultiTenantSaaS {  
  # Tenant configuration  
  tenant {  
    isolation: schema # or: database, row  
    routing: subdomain  
    customization: {  
      branding: true  
      fields: true  
    }  
  }  
}  
  
# Tenant-aware entity  
entity Document {  
  tenant: Tenant  
  name: string  
  content: text  
  
  access: tenant.users  
}  
}
```

Complex Workflows

```
app ApprovalSystem {  
  # State machine with parallel tracks  
  entity Request {  
    status: {  
      main: draft->submitted->approved,  
      finance: pending->reviewed->cleared,  
      legal: waiting->checked->signed  
    }  
  }  
}
```

```

rules {
  approve: {
    require: [
      finance.cleared,
      legal.signed,
      role.manager
    ]
  }
}
}
}

```

Plugin Architecture

```

app PluggableSystem {
  # Plugin system
  plugins {
    types: [ui, logic, data]
    isolation: process
    permissions: restricted
  }

  # Plugin hooks
  hooks {
    beforeCreate: [validate, enrich]
    afterUpdate: [notify, sync]
    onError: [log, retry]
  }
}

```

Event Sourcing

```

app EventSourced {
  # Event definitions
  events {
    OrderCreated: {
      order_id: id
      items: [Product]
      total: money
    }

    OrderShipped: {
      order_id: id
      tracking: string
      carrier: enum
    }
  }

  events {
    order.created: [
      notify@customer,

```



```

        update@inventory,
        track@analytics
    ]

    order.shipped: {
        handler: [
            update@status,
            send@tracking,
            notify@customer
        ]
        retry: {
            attempts: 3
            backoff: exponential
        }
    }
}

```

AI Integration

```

app AIEnhanced {
    # AI field types
    entity Content {
        text: ai.text {
            analyze: sentiment
            extract: [topics, entities]
            suggest: tags
        }

        image: ai.image {
            detect: [objects, faces, text]
            moderate: inappropriate
            enhance: quality
        }
    }

    # AI-powered rules
    rules {
        categorize: {
            using: ai.classifier
            model: content-type
            threshold: 0.8
        }
    }
}

```

Time-Based Patterns

```

app TimeAware {
    # Scheduling
    schedule {

```

```

daily: [
  cleanupTemp,
  updateStats
]
weekly: generateReport
custom: "0 9 * * 1-5" # Cron
}

# Temporal rules
entity Contract {
  valid: daterange

  rules {
    approve: {
      require: within@businessHours
      deadline: 2@businessDays
    }
  }
}
}

```

Best Practices

1. Plugin Design

- Use clear interfaces
- Implement security boundaries
- Handle failures gracefully
- Version plugin APIs

2. Event Sourcing

- Keep events immutable
- Use meaningful event names
- Include all relevant data
- Handle idempotency

3. AI Integration

- Set confidence thresholds
- Handle AI failures
- Respect privacy concerns
- Monitor AI performance

4. Time Management

- Consider timezones
 - Handle business calendars
 - Plan for failures
 - Monitor long-running tasks
-

Chapter 44

docs/reference/cli.md

Chapter 45

Command Line Interface

SeedML CLI focuses on simplicity with smart defaults. Most commands require minimal configuration.

Quick Start

```
// Generate app from spec (uses all smart defaults)
seedml create myapp.seed

// Run the generated app
cd myapp
seedml run
```

Essential Commands

```
seedml create <spec>      # Generate new app from spec(s)
seedml extend <app>       # Add components to existing app
seedml run                # Run the app locally
seedml deploy             # Deploy to production
seedml test               # Run all tests
```

Location Commands

```
seedml geocode <file>     # Batch geocode addresses
seedml regions verify     # Verify region definitions
seedml maps cache clear   # Clear geocoding cache
seedml maps config test   # Test maps configuration
```

Maps Configuration

Configure maps features through command line options or environment variables:

```
// Provider Selection
--maps-provider google|mapbox|osm // Select maps provider
--maps-version latest|legacy      // API version
```

```
// Geocoding Options
--geocoding-cache true|false      // Enable caching
--geocoding-precision high|normal // Coordinate precision
--geocoding-regions [codes]      // Restrict to regions

// Performance
--maps-clustering auto|manual    // Clustering strategy
--maps-cache-size 100MB          // Cache size limit
--maps-rate-limit 100/min        // API rate limit
```

Configuration

Configuration uses smart defaults - override only when needed:

```
// Environment (optional)
export SEEDML_ENV=dev           // dev/staging/prod
export ANTHROPIC_API_KEY=xxx    // For AI features
export GOOGLE_MAPS_KEY=xxx     // For maps features
export MAPS_CACHE_DIR=./cache  // For geocoding cache

// Command options (all optional)
seedml create myapp.seed \
  --stack modern                // Use latest tech stack
  --db postgres                 // Override default DB
  --port 3000                   // Override default port
  --maps-provider google        // Maps provider
  --geocoding-strategy cached    // Geocoding approach
```

Generated Stack

The CLI generates a complete, production-ready application with:

- Modern frontend (React + TypeScript)
- API backend (FastAPI)
- Database (PostgreSQL)
- Authentication
- Testing
- Documentation
- Deployment configs
- Maps components and services
- Geocoding pipeline
- Location caching system

All components use battle-tested patterns and best practices by default.

Learn More

- [Quick Start Guide](#)
- [Configuration Guide](#)
- [Deployment Guide](#)

Chapter 46

docs/reference/patterns.md

Chapter 47

Application Patterns

SeedML provides built-in patterns organized by architectural layer. These patterns ensure consistency and best practices across applications.

Foundation Layer Patterns

```
# Base entity pattern
entity BaseEntity {
  id: uuid
  created_at: timestamp = now()
  created_by: User
  updated_at: timestamp
  updated_by: User
  version: int = 1
}

# Common validation patterns
validation {
  required: field != null
  format: matches(pattern)
  range: between(min, max)
  money: positive and precision(2)
  quantity: positive and <= stock
}

# Error handling patterns
error_handling {
  retry: {
    attempts: 3
    backoff: exponential
    notify: on_final_failure
  }
  validation: {
    collect_all: true
    format: standard_error
  }
}
```

1. Foundation Patterns

```
# Type patterns
types {
  Email: string { format: email }
  Money: number { precision: 2 }
  Status: enum(active, inactive)
}

# Validation patterns
validate {
  required: field != null
  format: matches(pattern)
  range: between(min, max)
}
```

2. Data Patterns

```
# Entity patterns
entity Base {
  id: uuid
  created: timestamp
  updated: timestamp
}

# Relationship patterns
relationships {
  one_to_many: [parent->children]
  many_to_many: [products<->categories]
}
```

3. Logic Patterns

```
# Business rule patterns
rules {
  validation: require(condition)
  workflow: state->next_state
  computation: derived = formula
}
```

4. Security Patterns

```
# Permission patterns
permissions {
  entity_level: {
    entity: Type
    actions: [create, read, update]
  }
}
```



```
# Role patterns
roles {
  hierarchical: [admin->manager->user]
  functional: [billing, support, sales]
}
```

5. Presentation Patterns

```
# UI patterns
screens {
  layouts: [list, detail, dashboard]
  components: [table, form, chart]
  navigation: [menu, tabs, breadcrumbs]
}
```

6. Integration Patterns

```
# External service patterns
integrate {
  apis: [rest, graphql]
  events: [webhook, queue]
  sync: [batch, realtime]
}
```

CRUD Operations

```
# Basic CRUD pattern
entity Product {
  name: string
  price: money
  status: active/inactive = active
}

screen Products {
  list: [name, price, status]
  search: [name, status]
  actions: [create, edit, delete]
}
```

Workflow Management

```
# State machine workflow
entity Order {
  status: draft->submitted->approved->shipped

  rules {
    submit: {
      require: [items.valid, total > 0]
      then: notify@manager
    }
  }
}
```

```

    }
    approve: {
      require: role.manager
      then: [create@invoice, notify@customer]
    }
  }
}

```

Dashboard Layouts

```

screen Dashboard {
  layout: grid(2x2)

  widgets: [
    {
      type: counter
      data: Orders.count(today)
      compare: yesterday
    },
    {
      type: chart
      data: Sales.by(month)
      range: last_6_months
    }
  ]
}

```

Form Handling

```

screen OrderForm {
  form: {
    sections: [
      customer: [name!, email!, phone?],
      items: editable_table,
      notes: textarea
    ]

    validation: inline
    actions: [save_draft, submit]
  }
}

```

Search and Filter

```

screen Products {
  search: {
    quick: [name, sku]
    filters: {
      category: select,
      price: range,

```

```

    status: multiple
  }
  sort: [name, -created]
}
}

```

Access Control

```

app Portal {
  roles: [admin, manager, user]

  entity Document {
    access: {
      view: authenticated
      edit: role.manager
      delete: role.admin
    }
  }
}

```

Integration Patterns

```

integrate {
  # Event-driven integration
  stripe: {
    on: payment.success
    then: [approve@order, notify@customer]
  }

  # API integration
  weather: {
    url: "https://api.weather.com"
    cache: 30min
    retry: 3
  }
}

```

Multi-tenant Patterns

```

app SaaS {
  tenant {
    isolation: schema # or: database, row
    routing: subdomain

    customize: {
      branding: true
      fields: true
    }
  }
}

```

```
entity Document {  
  tenant: Tenant  
  access: tenant.users  
}  
}
```

Best Practices

1. Pattern Selection

- Use built-in patterns when possible
- Customize only when needed
- Maintain consistency across app

2. Pattern Composition

- Combine patterns effectively
- Keep patterns focused
- Document custom patterns

3. Pattern Evolution

- Start simple
 - Add complexity gradually
 - Refactor when needed
-

Chapter 48

docs/reference/types.md

Chapter 49

Type System Reference

Core Types

```
# Simplified primitive types
types {
  string          # Text values
  number          # Numeric values
  bool            # True/false values
  date            # Simple dates
}

# Common domain types
types {
  email           # Email addresses
  phone           # Phone numbers
  money           # Currency values
}

# Simple collections
[Type]           # Lists of values
Type             # References to entities

# Optional/Required modifiers
Type?            # Optional value
Type = default   # Default value
```

Basic Validation

```
# Simple validation rules
string {
  validate: {
    required: bool    # Field is required
    unique: bool      # Values must be unique
    min: number       # Minimum length
    max: number       # Maximum length
  }
}
```

```

number {
  min: number      # Minimum value
  max: number      # Maximum value
}

# Format validation
email: string      # Validates email format
phone: string      # Validates phone format
money: number      # Validates currency format

```

Type Inference

SeedML automatically infers types in many contexts:

```

entity Product {
  price: 0.00      # Inferred: money
  created: now()   # Inferred: timestamp
  active: true     # Inferred: boolean
  tags: []        # Inferred: [string]
}

```

Custom Types

Define reusable custom types:

```

types {
  # Simple custom type
  Currency: money {
    precision: 2
    positive: true
  }

  # Complex custom type
  Address: {
    street: string!
    city: string!
    state: enum(...)
    zip: string {
      pattern: "\\d{5}"
    }
  }

  # Enum type
  Status: enum(
    active: "Active",
    inactive: "Inactive",
    pending: "Pending Review"
  )
}

```

Type Composition

Build complex types through composition:

```
entity Order {  
  # Nested structure  
  shipping: {  
    address: Address  
    method: ShippingMethod  
    tracking?: string  
  }  
  
  # List of complex items  
  items: [{  
    product: Product  
    quantity: int > 0  
    price: Currency  
  }]  
}
```

Best Practices

1. **Type Selection**
 - Use specific types over generic ones
 - Consider validation requirements
 - Think about UI rendering
 - Plan for future needs
2. **Validation**
 - Add constraints appropriately
 - Use built-in validations
 - Create reusable types
 - Document custom types
3. **Performance**
 - Consider database implications
 - Plan indexes carefully
 - Use appropriate list types
 - Monitor complex types
4. **Maintenance**
 - Document custom types
 - Use consistent patterns
 - Plan for versioning
 - Consider migrations

Basic Structure

Every SeedML application follows this structure:

```
app [AppName] {  
  # Global Configuration  
  meta: { ... }  
  
  # Data Models  
  entity [EntityName] { ... }
```



```
# Business Rules
rules { ... }

# User Interface
screens { ... }

# Integrations
integrate { ... }
}
```

Entity Syntax

```
entity [EntityName] {
  # Basic fields
  name: string
  age: number
  active: bool = true

  # Field modifiers
  required: string!
  optional: string?
  defaulted: string = "default"

  # Complex types
  items: [Item]
  metadata: map<string,any>
  status: draft->submitted->approved
}
```

Chapter 50

docs/examples/basic-crud.md

Chapter 51

Basic CRUD Example

The simplest possible SeedML application showing smart defaults in action.

```
// core.seed - Core domain model
app Contacts {
  entity Contact {
    // Required fields
    name: string!           // Full name
    email: email!           // Email with validation

    // Optional fields with validation
    phone?: phone {         // Phone number
      format: international
    }
  }
}

// ui.seed - UI components
extend Contacts {
  // Theme configuration
  ui {
    theme: "light" // Use built-in theme
  }

  location?: location { // Location with defaults
    validate: {
      region: service_area
    }
  }

  // Metadata
  created: timestamp = now()
  updated: timestamp
  version: int = 1
}

// Complete UI with maps
screen Contacts {
  views: {
    list: {
```

```

        fields: [name, email, phone],
        actions: [create, edit, delete],
        search: [name, email],
        sort: name
    },
    map: {
        source: location,
        cluster: true,
        search: {
            radius: 10km,
            filters: [type, status]
        }
    }
}
}
}
}

```

What You Get

This minimal specification automatically generates:

Data Layer

- Database schema with proper types and indexes
- Input validation for all fields
- API endpoints for CRUD operations
- Search and filter capabilities

User Interface

- Responsive list/grid view
- Create and edit forms
- Search functionality
- Sort and filter options
- Mobile friendly layout

Features

- Authentication and authorization
- Error handling
- Success messages
- Audit logging
- API documentation

Map Features

- Interactive map view
- Location picker for editing
- Address autocomplete
- Distance calculations
- Clustering for multiple contacts
- Mobile-friendly controls

All of these features come from smart defaults - no additional configuration needed.

Progressive Enhancement

When you need customization:

```
app Contacts {
  // Override specific defaults
  entity Contact {
    name: string {
      min: 2,           # Min length
      max: 50,          # Max length
      case: title       # Title case
    }
    email: email        # Keep email defaults
    phone: phone?       # Keep phone defaults
    location: location {
      required: true
      validate: {
        region: service_area
        type: business
      }
    }
  }
}

// Customize UI
screen Contacts {
  // Enhanced list view
  list {
    show: [name, email, location]
    group: region
    sort: distance(current_location)
  }

  // Enhanced map view
  map {
    cluster: true
    search: {
      radius: 10km
      filters: [region, type]
    }
    interactions: [
      select: show_details,
      route: get_directions
    ]
  }
}
```

Key Principles Demonstrated

1. **Minimal Valid Specification:** Express only what's unique about your application
2. **Smart Defaults:** Production patterns included automatically
3. **Progressive Enhancement:** Add complexity only when needed
4. **Intent-Focused:** Express what you want, not how to build it
5. **Location-Aware:** Seamless integration of mapping features

Chapter 52

docs/examples/business-app.md

Chapter 53

Business Application Example

A complete order management system showing how intent-focused patterns scale to larger applications.

```
# core.seed - Core domain model
app OrderSystem {
  # Theme configuration
  ui {
    theme: "modern" # Use modern theme
  }

  # 1. Domain Model
  entity Customer {
    # Basic information
    name: string!
    email: email!
    phone: phone?
    status: active/inactive = active

    # Financial & verification
    verified: boolean = false
    creditLimit: money = 1000.00

    # Validation rules
    validate: {
      email: required if status == active
      creditLimit: positive
    }

    # Customer verification process
    rules {
      verify: {
        require: [
          email != null,
          status == active
        ]
        then: [
          set(verified, true),
          notify@customer
        ]
      }
    }
  }
}
```



```

    }
  }
}

entity Product {
  name: string!
  description: text
  price: money!
  inStock: boolean = true
  stockQuantity: int > 0

  validate: {
    price: positive
    stockQuantity: positive
  }
}

# Dependent entities
entity OrderItem {
  product: Product!
  quantity: int > 0
  price: money!
  total: quantity * price

  validate: {
    quantity: <= product.stockQuantity
    price: == product.price
  }
}

entity Order {
  # Basic fields
  customer: Customer!
  items: [OrderItem]
  status: draft->submitted->approved->shipped

  # Computed fields
  subtotal: sum(items.total)
  tax: subtotal * 0.2
  total: subtotal + tax

  # Business rules
  rules {
    submit: {
      require: [
        items.length > 0,
        customer.verified,
        total <= customer.creditLimit
      ]
      error: {
        items.length: "Order must contain at least one item"
        customer.verified: "Customer must be verified before placing orders"
        creditLimit: "Order total exceeds customer credit limit"
      }
    }
  }
}

```

```

    then: notify@sales
  }

  approve: {
    require: role.manager
    check: items.all(product.inStock)
    error: {
      role: "Only managers can approve orders"
      stock: "Some items are out of stock"
    }
    then: [
      create@invoice,
      notify@warehouse
    ]
  }

  ship: {
    require: status == approved
    error: {
      status: "Order must be approved before shipping"
    }
    then: [
      update@inventory,
      notify@customer,
      notify@shipping
    ]
  }
}

entity Invoice {
  order: Order!
  issueDate: datetime = now()
  dueDate: datetime = issueDate + 30.days
  status: pending/paid/overdue = pending
  amount: money = order.total
}

# Permissions (now can reference known entities)
permissions {
  # Customer-related permissions
  view_all_customers: {
    entity: Customer
    access: read
    filter: all
  }
  view_active_customers: {
    entity: Customer
    access: read
    filter: status == active
  }
  set_credit_limits: {
    entity: Customer
    access: [read, update]
  }
}

```

```

    fields: [creditLimit]
    validate: {
      creditLimit: <= 50000
    }
  }

# Order-related permissions
create_orders: {
  entity: Order
  access: create
  fields: [customer, items]
}
view_own_orders: {
  entity: Order
  access: read
  filter: created_by == current_user
}
view_customer_orders: {
  entity: Order
  access: read
  filter: customer.id == current_user.id
}
approve_orders: {
  entity: Order
  access: [read, update]
  filter: status == submitted
  allow: [approve]
}
submit_orders: {
  entity: Order
  access: [read, update]
  filter: created_by == current_user
  allow: [submit]
}

# Product-related permissions
manage_products: {
  entity: Product
  access: [create, read, update]
  filter: all
}

# Roles now reference permissions
roles {
  admin: {
    permissions: [all]
  }

  manager: {
    permissions: [
      view_all_customers,
      approve_orders,
      manage_products,

```

```

        set_credit_limits
    ]
}

sales: {
  permissions: [
    view_active_customers,
    create_orders,
    submit_orders,
    view_own_orders
  ]
}

customer: {
  permissions: [
    view_customer_orders,
    create_orders
  ]
}
}
# Customer entity definition
entity Customer {
  # Basic information
  name: string!
  email: email!
  phone: phone?
  status: active/inactive = active

  # Financial & verification
  verified: boolean = false
  creditLimit: money = 1000.00

  # Validation rules
  validate: {
    email: required if status == active
    creditLimit: positive
  }

  # Customer verification process
  rules {
    verify: {
      require: [
        email != null,
        status == active
      ]
      then: [
        set(verified, true),
        notify@customer
      ]
    }
  }
}

# Product entity definition

```

```

entity Product {
  name: string!
  description: text
  price: money!
  inStock: boolean = true
  stockQuantity: int > 0

  validate: {
    price: positive
    stockQuantity: positive
  }
}

# Order entity definition
entity Order {
  # Basic fields
  customer: Customer!
  items: [OrderItem]
  status: draft->submitted->approved->shipped

  # Computed fields
  subtotal: sum(items.total)
  tax: subtotal * 0.2
  total: subtotal + tax

  # Business rules
  rules {
    submit: {
      require: [
        items.length > 0,
        customer.verified,
        total <= customer.creditLimit
      ]
      error: {
        items.length: "Order must contain at least one item"
        customer.verified: "Customer must be verified before placing orders"
        creditLimit: "Order total exceeds customer credit limit"
      }
      then: notify@sales
    }

    approve: {
      require: role.manager
      check: items.all(product.inStock)
      error: {
        role: "Only managers can approve orders"
        stock: "Some items are out of stock"
      }
      then: [
        create@invoice,
        notify@warehouse
      ]
    }
  }
}

```

```

    ship: {
      require: status == approved
      error: {
        status: "Order must be approved before shipping"
      }
      then: [
        update@inventory,
        notify@customer,
        notify@shipping
      ]
    }
  }
}

```

```

entity OrderItem {
  product: Product!
  quantity: int > 0
  price: money!
  total: quantity * price

  validate: {
    quantity: <= product.stockQuantity
    price: == product.price
  }
}

```

```

entity Invoice {
  order: Order!
  issueDate: datetime = now()
  dueDate: datetime = issueDate + 30.days
  status: pending/paid/overdue = pending
  amount: money = order.total
}

```

UI Screens (reference entities and roles)

```

screen Products {
  list {
    view: table
    show: [name, price, inStock, stockQuantity]
    actions: [create, edit]
  }
}

```

```

screen Customers {
  list {
    view: table
    show: [name, email, status, verified, creditLimit]
    actions: [create, edit, verify]
  }
}

```

```

detail {
  layout: tabs
  info: [name, email, phone, status]
  orders: related-list(Order)
}

```

```

    invoices: related-list(Invoice)
  }
}

screen Orders {
  list {
    view: table
    show: [id, customer.name, total, status]
    group: status
    actions: [
      submit if draft,
      approve if submitted and role.manager,
      ship if approved
    ]
  }

  detail {
    layout: split
    left {
      customer: card {
        show: [name, email, creditLimit, verified]
      }
      items: editable-table {
        columns: [product.name, quantity, price, total]
        validate: onEdit
      }
    }
    right {
      summary: [subtotal, tax, total]
      status: timeline
      actions: panel
    }
  }
}

# External integrations (last since they may reference entities and actions)
integrate {
  payment: stripe      # Payment processing
  shipping: fedex       # Shipping integration
  email: sendgrid       # Email service

  config {
    stripe_key: env.STRIPE_KEY
    fedex_account: env.FEDEX_ACCOUNT
    templates: {
      order: "d-123456"
      shipping: "d-789012"
    }
  }
}

```

This example demonstrates: - Complete entity definitions with relationships - Comprehensive business rules and validation - Computed fields and dependencies - Multi-step workflows with proper states - Rich UI components and layouts -

Chapter 54

docs/examples/dashboard.md

Chapter 55

Analytics Dashboard Example

```
// Real-time analytics dashboard with location intelligence
app Analytics {
  // Data models
  entity Metric {
    name: string
    value: number
    timestamp: datetime
    category: string
    location?: location    // Optional location data
  }

  entity LocationMetric {
    location: location
    metrics: map<string, number>
    timestamp: datetime
    region: reference
  }

  # Dashboard screen
  screen Dashboard {
    layout: grid(2x2)

    // Multiple visualization types
    widgets: [
      {
        type: map
        data: LocationMetric
        view: {
          type: heatmap
          cluster: {
            enabled: true
            threshold: 100
            radius: 50
            colors: gradient
          }
        }
        controls: {
          zoom: true
        }
      }
    ]
  }
}
```

```

        pan: true
        search: {
            radius: 10km
            filters: [type, status]
        }
    }
    value: {
        field: metrics.value
        range: last-7-days
        aggregation: sum
    }
},
{
    type: region-map
    data: Metric.by(region)
    color: value
    legend: true
    interact: drill_down
},
{
    type: line-chart
    data: Metric
    x: timestamp
    y: value
    group: category
    range: last-7-days
},
{
    type: cluster-map
    data: EventMetric
    cluster: category
    size: value
    tooltip: details
}
]

// Interactive features
features: [
    time-range-selector,
    region-filter,
    category-filter,
    export-data
]

# Real-time updates
refresh: 5min
realtime: websocket
}

# Location analytics
analytics {
    spatial: {

```

```

hotspots: {
  method: kernel_density
  threshold: significant
  timeframe: hourly
},
patterns: {
  type: movement
  window: 24h
  detect: [clusters, flows]
}
}

metrics: {
  density: per_km2
  distribution: by_region
  concentration: gini_index
}
}

# Map controls
controls {
  view: {
    type: [heat, cluster, region]
    switch: animated
    sync: linked
  },

  analysis: {
    tools: [
      area_selection,
      pattern_detection
    ],
    export: [
      geojson,
      csv,
      image
    ]
  }
}
}

```

This example demonstrates: - Location-based analytics - Spatial visualization - Real-time updates - Interactive mapping
 - Multiple chart types - Pattern detection - Integrated dashboard layout

Chapter 56

docs/examples/integration.md

Chapter 57

Integration Examples

This guide shows how to integrate external services using SeedML's simplified patterns.

Basic Integration

```
app Store {
  # Simple integrations
  integrate {
    auth: {
      provider: google    # Basic auth
      features: default
    }
    email: {
      provider: sendgrid  # Email service
      features: default
    }
    storage: {
      provider: s3        # File storage
      features: default
    }
  }

  # Use integrations naturally
  entity Order {
    status: Draft -> Paid -> Shipped
    attachment: file    # Uses storage

    rules {
      create: {
        then: notify@customer # Uses email
      }
    }
  }
}
```

Common Patterns

1. Authentication

```
integrate {  
  auth: {  
    provider: google    # Single provider  
    redirect: "/home"  # After login  
  }  
}
```

2. File Storage

```
integrate {  
  storage: {  
    provider: s3  
    bucket: "app-files" # Single bucket  
  }  
}
```

3. Email Service

```
integrate {  
  email: {  
    provider: sendgrid  
    from: "app@example.com"  
  }  
}
```

Best Practices

1. Keep It Simple

- Use single providers
- Minimal configuration
- Default behaviors

2. Security First

- Environment variables for keys
- HTTPS connections
- Basic access control

3. Handle Errors

- Basic retry logic
- Simple error states
- Clear messages

Smart Defaults

Every integration includes: - Basic error handling - Simple retry logic - Standard logging - Essential security

Chapter 58

docs/examples/saas.md

Chapter 59

SaaS Application Example

```
# Multi-tenant SaaS platform
app SaaSPlatform {
  # Tenant configuration
  tenant {
    isolation: schema # database, schema, or row
    routing: subdomain

    customize: {
      branding: {
        logo: image
        colors: theme
        domain: url?
      }
      features: {
        enabled: [module]
        limits: map<feature, limit>
      }
    }
  }
}

# Subscription management
subscription {
  plans: {
    basic: {
      price: 10/month
      limits: {
        users: 5
        storage: 5gb
        api: 1000/day
      }
    }
  }

  pro: {
    price: 50/month
    limits: {
      users: 50
      storage: 50gb
    }
  }
}
```

```

        api: 10000/day
    }
}

enterprise: {
    price: custom
    limits: custom
}

features: {
    basic: [core, support],
    pro: [api, advanced, priority],
    enterprise: [custom, sla, training]
}

# Cross-tenant features
cross_tenant {
    sharing: {
        content: {
            access: explicit
            audit: true
        }

        marketplace: {
            publish: verified
            install: compatible
        }
    }

    analytics: {
        usage: aggregate
        trends: anonymous
    }
}

# Tenant management
admin {
    dashboard: {
        metrics: [
            active_tenants,
            total_users,
            storage_used,
            api_usage
        ]

        actions: [
            provision,
            suspend,
            migrate,
            backup
        ]
    }
}

```

```

    }

    monitoring: {
      health: [status, performance],
      alerts: [limits, errors],
      audit: [access, changes]
    }
  }

  # Tenant-aware entities
  entity Document {
    tenant: Tenant
    sharing: private/shared/public

    # Tenant isolation
    access: tenant.users
    storage: tenant.bucket
    audit: tenant.log
  }

  # Tenant-specific UI
  screen Dashboard {
    branding: tenant.theme
    modules: tenant.enabled
    limits: tenant.quotas

    widgets: [
      usage: tenant.metrics,
      users: tenant.members,
      activity: tenant.events
    ]
  }
}

```

This example demonstrates: - Multi-tenant architecture - Tenant isolation - Custom branding - Feature management - Usage tracking - Cross-tenant features

Chapter 60

.pytest_cache/README.md

Chapter 61

pytest cache directory

This directory contains data from the pytest's cache plugin, which provides the `--lf` and `--ff` options, as well as the cache fixture.

Do not commit this to version control.

See [the docs](#) for more information.
