Colors of Life - Project Rules

This document defines the coding standards, development practices, and organizational rules for the Colors of Life project. Following these guidelines ensures consistency, quality, and maintainability across the codebase.

Code Structure and Organization

Directory Structure

- Follow the structure defined in the File Structure document
- Each service should maintain a consistent internal structure
- Group related functionality into modules or packages
- Keep file size manageable (< 500 lines recommended)

Naming Conventions

General

- Use descriptive, intention-revealing names
- Avoid abbreviations except for commonly accepted ones
- Maintain consistent casing conventions per language

TypeScript/JavaScript

- **Files**: kebab-case for filenames (e.g., `user-service.ts`)
- **Components**: PascalCase (e.g., `ProductCard.tsx`)
- **Functions**: camelCase (e.g., `getUserProfile`)
- **Constants**: UPPER_SNAKE_CASE (e.g., `MAX_RETRY_ATTEMPTS`)
- **Interfaces/Types**: PascalCase with prefix I for interfaces (e.g., `IUserProfile`, `ProductData`)
- **CSS Modules**: kebab-case (e.g., `product-card.module.css`)

Go

- **Files**: snake_case (e.g., `user_service.go`)
- **Functions**: PascalCase for exported, camelCase for internal
- **Interfaces**: PascalCase (e.g., `UserRepository`)
- **Constants**: PascalCase for exported, camelCase for internal

Rust

- **Files**: snake_case (e.g., `try_on_service.rs`)
- **Functions**: snake_case (e.g., `process_image`)
- **Structs/Enums**: PascalCase (e.g., `TryOnRequest`)
- **Constants**: UPPER_SNAKE_CASE (e.g., `MAX_IMAGE_SIZE`)

Import Order

- Group imports in the following order:
  1. Standard library imports
  2. Third-party library imports
  3. Internal module imports
  4. Relative imports
- Sort alphabetically within each group
- Add a blank line between groups

typescript Copy

```typescript
// Example import order in TypeScript
import { useEffect, useState } from 'react';

import { clsx } from 'clsx';
import { motion } from 'framer-motion';
import { useQuery } from 'react-query';

import { useAuthStore } from '@/store/auth';
import type { Product } from '@/types';

import { ProductDescription } from './ProductDescription';
import styles from './product-card.module.css';
```

Coding Standards

General Principles

- Write self-documenting code
- Follow the DRY (Don't Repeat Yourself) principle
- Keep functions small and focused on a single responsibility

- Prioritize readability over cleverness
- Use meaningful variable names that reveal intent

Language-Specific Guidelines

TypeScript/JavaScript

- Use TypeScript for all new code
- Enable strict mode
- Define explicit return types for functions
- Use interfaces for object shapes
- Use enums for fixed sets of values
- Prefer functional programming patterns
- Use optional chaining and nullish coalescing
- Prefer async/await over Promise chains

Go

- Follow the Effective Go guidelines
- Use meaningful error messages
- Implement proper error handling
- Use context for request-scoped values
- Follow standard project layout

Rust

- Follow the Rust API Guidelines
- Use Result and Option types appropriately
- Leverage the type system for safety
- Write comprehensive documentation
- Use idiomatic Rust patterns

Testing Guidelines

- Write tests before or alongside production code
- Test both success and error paths
- Keep tests independent and idempotent

- Use descriptive test names that explain the expected behavior
- Structure tests using Arrange-Act-Assert pattern
- Use mocks and stubs sparingly and explicitly

Comments and Documentation

- Document "why" not "what" or "how"
- Use JSDoc/TSDoc for public APIs and complex functions
- Keep comments up-to-date with code changes
- Write module-level documentation explaining purpose and usage
- Document non-obvious decisions and edge cases

Version Control Practices

Branching Strategy

- Main branch contains production-ready code
- Feature branches for new development
- Release branches for version preparation
- Hotfix branches for urgent production fixes

Commit Guidelines

- Write meaningful commit messages
- Follow conventional commits format:
  - `feat`: New feature
  - `fix`: Bug fix
  - `docs`: Documentation changes
  - `style`: Formatting changes
  - `refactor`: Code change that neither fixes a bug nor adds a feature
  - `test`: Adding or updating tests
  - `chore`: Changes to build process or auxiliary tools
- Example: `feat(try-on): implement Kling API integration`
- Keep commits focused and atomic
- Rebase feature branches before merging

Pull Request Process

- Create descriptive PR titles
- Fill out the PR template completely
- Link related issues in the PR description
- Keep PRs focused on a single change
- Request reviews from appropriate team members
- Address all review comments
- Ensure all checks pass before merging
- Squash commits when merging to main

Development Workflow

Issue Tracking

- All work should be associated with an issue
- Use issue templates for different types of work
- Label issues appropriately (bug, feature, enhancement, etc.)
- Assign priority and milestone to each issue
- Link PRs to issues they address

Sprint Process

- Two-week sprint cycles
- Sprint planning at the beginning of each sprint
- Daily standup meetings
- Sprint review and retrospective at the end
- Maintain a prioritized backlog

Definition of Ready

An issue is ready for development when it:

- Has clear acceptance criteria
- Is properly sized and estimated
- Has all necessary design specs
- Dependencies are identified

- Necessary resources are available

Definition of Done

A feature is considered done when:

- Code is written and tested
- Documentation is updated
- Code is reviewed and approved
- Integration tests pass
- Acceptance criteria are met
- No regressions are introduced
- Feature is deployable to production

Code Quality Standards

Linting and Formatting

- Use ESLint for TypeScript/JavaScript
- Use gofmt for Go
- Use Rust formatting tools
- Run linters in CI pipeline
- Fix all linting issues before merging

Static Analysis

- Run static analysis tools as part of CI
- Address all critical and high-priority issues
- Gradually improve quality score over time

Code Review Guidelines

- Review for correctness, readability, and maintainability
- Check for security vulnerabilities
- Verify error handling
- Ensure test coverage
- Look for performance issues
- Provide constructive feedback

- Approve only when all concerns are addressed

Performance Standards

- Page load time < 3 seconds
- API response time < 500ms for standard requests
- Try-on processing feedback within 5 seconds
- Mobile-first, responsive design
- Optimize for low-bandwidth scenarios

Security Requirements

Authentication & Authorization

- Use industry-standard authentication protocols
- Implement proper role-based access control
- Generate strong, random session identifiers
- Set secure cookie attributes
- Implement proper logout mechanisms

Data Protection

- Encrypt sensitive data at rest and in transit
- Use parameterized queries to prevent SQL injection
- Sanitize all user inputs
- Implement proper content security policy
- Follow least privilege principle

API Security

- Validate all inputs
- Rate limit API endpoints
- Use secure token validation
- Implement proper CORS policy
- Log security-related events

User Data Privacy

- Collect only necessary data

- Provide clear privacy notices
- Implement data retention policies
- Support data export and deletion
- Comply with relevant regulations (GDPR, CCPA)

Kling AI Integration Rules

API Usage

- Store API keys securely in environment variables or secrets manager
- Implement proper error handling for all API calls
- Follow rate limiting guidelines provided by Kling
- Cache results when appropriate
- Log all API interactions for troubleshooting

Image Handling

- Validate images meet Kling requirements before sending
- Optimize images for better processing
- Handle Base64 encoding/decoding properly
- Securely store user reference images
- Respect 30-day result expiration policy

Response Processing

- Implement robust error handling
- Provide meaningful error messages to users
- Store and track task IDs
- Implement proper polling strategy
- Cache successful results

Deployment and Operations

Environment Configuration

- Use environment variables for configuration
- Keep secrets out of code repositories
- Use different configurations for development, staging, and production

- Document all configuration options
- Validate configuration on startup

Logging Standards

- Use structured logging (JSON format)
- Include correlation IDs for request tracing
- Log appropriate information at each level
- Avoid logging sensitive information
- Include context information in logs

Monitoring Requirements

- Monitor system health metrics
- Track business KPIs
- Set up alerts for critical issues
- Monitor error rates and performance
- Use distributed tracing for request flows

Incident Response

- Define severity levels for incidents
- Document response procedures
- Establish on-call rotation
- Conduct post-mortems for significant incidents
- Implement lessons learned

Accessibility Standards

Requirements

- Comply with WCAG 2.1 AA standards
- Support keyboard navigation
- Ensure proper contrast ratios
- Provide text alternatives for non-text content
- Make interactive elements easily identifiable
- Support screen readers

- Design for various input methods

Testing

- Include accessibility in QA process
- Use automated accessibility testing tools
- Conduct manual testing with assistive technologies
- Address all accessibility issues before release

Performance Optimization

Frontend

- Minimize JavaScript bundle size
- Optimize images and assets
- Implement lazy loading
- Use code splitting
- Optimize critical rendering path
- Cache API responses appropriately

Backend

- Optimize database queries
- Implement appropriate caching
- Minimize network requests
- Use connection pooling
- Optimize resource utilization
- Implement proper pagination

Mobile Optimization

- Optimize for various network conditions
- Minimize battery usage
- Reduce memory footprint
- Support offline capabilities where possible
- Test on various device capabilities

Documentation Requirements

Code Documentation

- Document public APIs
- Explain complex algorithms
- Document non-obvious decisions
- Keep documentation up-to-date with code changes
- Use standard documentation formats

User Documentation

- Create comprehensive user guides
- Provide clear onboarding instructions
- Document feature usage
- Update documentation for each release
- Gather feedback on documentation clarity

Technical Documentation

- Maintain architecture diagrams
- Document system dependencies
- Create runbooks for operations
- Document API endpoints
- Provide setup instructions

Collaboration Guidelines

Communication Channels

- Use designated Slack channels for team communication
- Project management in Linear
- Technical documentation in Confluence
- Code discussions in GitHub/GitLab
- Daily standup meetings via Zoom

Meeting Practices

- Publish agenda before meetings
- Keep meetings focused and timeboxed

- Document decisions and action items
- Follow up on action items
- Respect everyone's time and input

Knowledge Sharing

- Schedule regular tech talks
- Document learnings and best practices
- Share articles and resources
- Cross-train team members
- Create a knowledge base for common issues

Third-Party Dependencies

Selection Criteria

- Evaluate license compatibility
- Check for active maintenance
- Consider community size and support
- Assess security track record
- Evaluate performance impact
- Test compatibility with existing stack

Management

- Document all dependencies with versions
- Regular updates for security patches
- Schedule major version upgrades
- Test thoroughly before upgrading
- Monitor for deprecated dependencies

UI/UX Standards

Design System

- Follow the Colors of Life design system
- Use standard components from the component library
- Maintain consistent spacing, typography, and colors

- Follow established interaction patterns
- Adhere to the "Apple-like" aesthetic defined in guidelines

Responsive Design

- Design mobile-first
- Support all major breakpoints
- Test on various screen sizes
- Ensure touch-friendly interactions
- Maintain usability across devices

Animation Guidelines

- Use subtle, purposeful animations
- Ensure animations can be disabled (prefers-reduced-motion)
- Maintain 60fps performance
- Use hardware-accelerated properties
- Keep animations under 300ms for UI feedback

Quality Assurance

Testing Requirements

- Unit tests for all services
- Integration tests for service interactions
- End-to-end tests for critical user flows
- Performance tests for key operations
- Accessibility testing
- Cross-browser compatibility testing

Bug Management

- Document all bugs with clear reproduction steps
- Classify by severity and priority
- Fix critical bugs before new feature development
- Regression test fixed bugs
- Track bug metrics to identify problem areas

Continuous Improvement

Code Refactoring

- Schedule regular refactoring sessions
- Address technical debt proactively
- Improve code quality incrementally
- Document refactoring decisions
- Test thoroughly after refactoring

Team Development

- Provide regular feedback
- Identify training opportunities
- Encourage innovation and experimentation
- Recognize and celebrate achievements
- Support career growth and development

Conclusion

These project rules provide a framework for consistent, high-quality development practices across the Colors of Life platform. They should evolve as the project progresses and the team learns what works best.

All team members are expected to follow these guidelines and suggest improvements when appropriate. Regular reviews of these rules should be conducted to ensure they remain relevant and effective.

Remember that the ultimate goal is to create an exceptional user experience while maintaining a sustainable and enjoyable development process. Rules should serve the project, not hinder progress or innovation.