Code-Factory System Architecture

Version: 1.0.0
Status: Draft
Last Updated: 2026-01-07
Owner: Code-Factory Core Team

Table of Contents

Executive Summary

Code-Factory is a **spec-driven software development system** that transforms natural language specifica-
tions into production-quality code through AI-powered workflows. The system is designed as a **single, zero-dependency Go binary** with an elegant terminal UI, supporting both local (Ollama) and cloud-based (OpenAI, Claude) LLM providers.

Key Characteristics

- **Distribution**: Single binary, no dependencies
- **Interface**: Beautiful TUI (Charm.sh/Bubble Tea) + optional web mirror
- **Storage**: Git-native flat files (markdown + reports)
- **AI**: Pluggable LLM backend (Ollama embedded or BYOK)
- **Integration**: Optional GitHub OAuth for seamless workflow
- **Target**: Individual developers and small teams

Design Goals

1. **Effortless**: Install and productive in 60 seconds
2. **Beautiful**: Terminal-first with aesthetic TUI
3. **Intelligent**: AI-powered but human-in-the-loop
4. **Collaborative**: Git-native, PR-ready outputs
5. **Flexible**: Works offline, online, with any LLM

Architecture Principles

1. Simplicity Over Complexity

Rationale: Developers want tools that “just work” without extensive setup.

Implementation:
- Single binary distribution (no package managers, no dependencies)
- Sensible defaults for everything
- Auto-detection and auto-configuration where possible
- Progressive disclosure of advanced features

2. Git-Native Storage

Rationale: Developers already know git; don’t introduce new data models.

Implementation:
- All specifications are markdown files in `/contracts/`
- All reports are markdown files in `/reports/`
- No databases, no proprietary formats
- Versionable, diffable, reviewable via standard git tools

3. Terminal-First Experience

Rationale: Developers live in terminals; GUI apps break flow.

Implementation:
- Primary interface is Charm.sh TUI (canvas-style, not line-by-line)
- Rich interactions: forms, menus, progress indicators
- Optional web UI as mirror for stakeholders
- Full keyboard navigation, no mouse required

4. Privacy & Security by Default

Rationale: Code is sensitive; developers value privacy.

Implementation:
- Prefer local Ollama over cloud LLMs
- Secrets in OS keyring, never in config files
- Explicit consent for any external communication
- Audit log of all AI interactions

5. Human-in-the-Loop

Rationale: AI is powerful but not infallible; humans must validate.

Implementation:
- All AI outputs require human review
- Clear diff views before any code changes
- Undo/rollback mechanisms
- Rescue mode for when AI goes off-rails

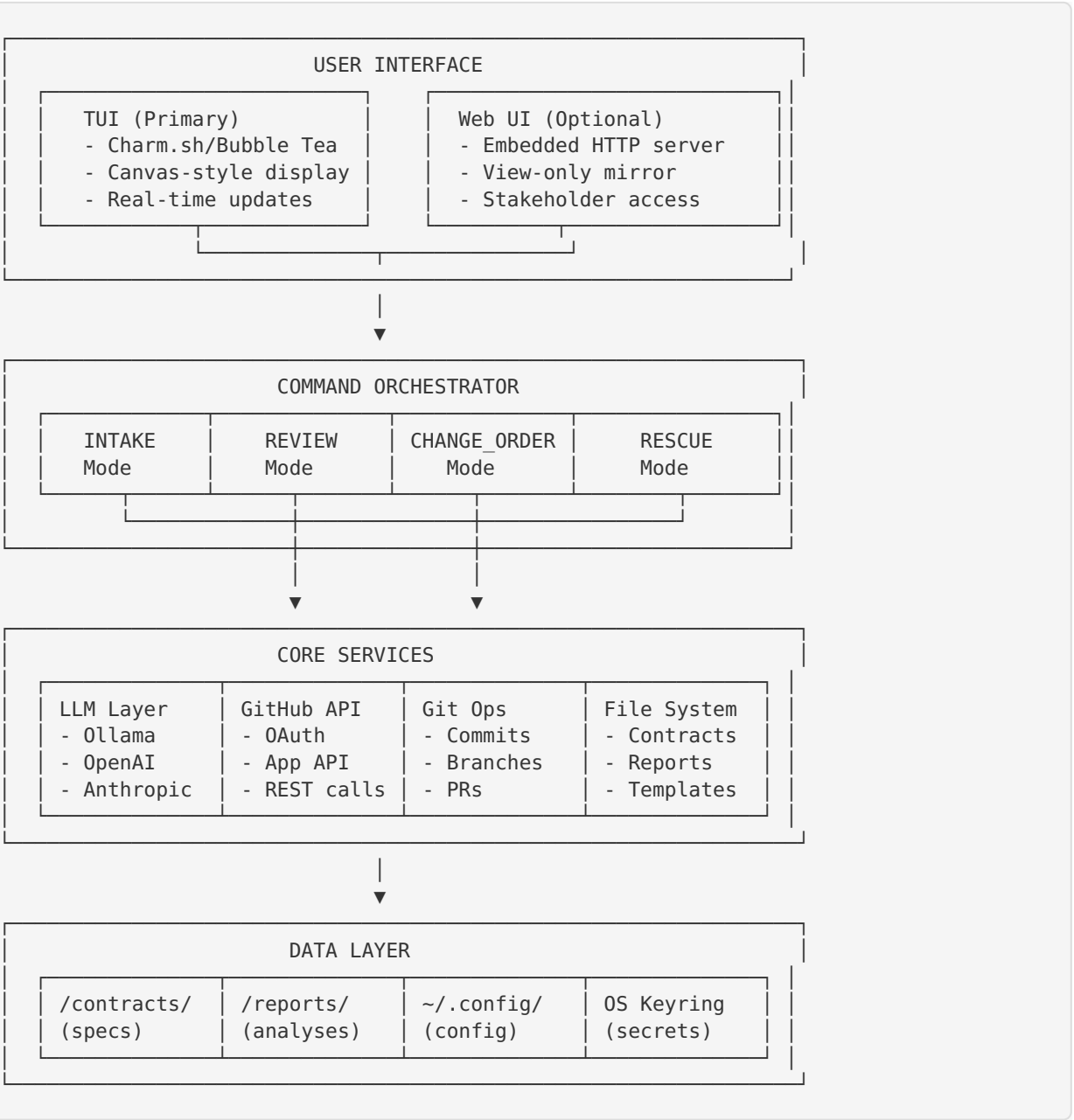
6. Extensibility Through Simplicity

Rationale: Advanced users want to customize without fighting the tool.

Implementation:
- Plain text files enable any editor/tool integration
- Plugin system via executable hooks (future)
- Configuration in YAML, scriptable with any language
- REST API for web UI enables custom clients

System Overview

High-Level Architecture



Component Layers

Layer 1: User Interface

- **TUI**: Primary interface for all operations
- **Web UI**: Optional read-only mirror for non-technical stakeholders

Layer 2: Command Orchestrator

- **Mode Handlers**: INTAKE, REVIEW, CHANGE_ORDER, RESCUE
- **Workflow Engine**: State machine driving each mode
- **Validation**: Pre/post checks for operations

Layer 3: Core Services

- **LLM Layer**: Abstraction over multiple LLM providers
- **GitHub API**: OAuth, App API, REST operations
- **Git Ops**: Local git operations via go-git
- **File System**: Safe file I/O with rollback

Layer 4: Data Layer

- **Contracts**: Markdown specs (git-tracked)
- **Reports**: Analysis outputs (git-tracked)
- **Config**: User preferences (YAML)
- **Secrets**: OS keyring integration

Component Architecture

1. TUI Component

Technology: Charm.sh Bubble Tea + Lipgloss + Bubbles

Architecture:

```
internal/tui/
├── app.go                  # Main TUI application
├── models/
│   ├── base.go             # Base model with common state
│   ├── intake.go           # INTAKE mode model
│   ├── review.go           # REVIEW mode model
│   ├── change_order.go     # CHANGE_ORDER mode model
│   └── rescue.go           # RESCUE mode model
├── components/
│   ├── header.go           # Top status bar
│   ├── footer.go           # Help/shortcuts bar
│   ├── sidebar.go          # Mode selector
│   ├── editor.go           # Spec editor widget
│   ├── diff_viewer.go      # Code diff display
│   ├── file_tree.go        # File browser
│   └── progress.go         # Progress indicators
├── styles/
│   └── theme.go            # Color schemes
└── utils/
    └── layout.go           # Layout calculations
```

Key Features:
- **Canvas-style rendering**: Full screen, not line-by-line
- **Real-time updates**: Progress bars, spinners, status updates
- **Keyboard-driven**: No mouse required (but supported)
- **Responsive layout**: Adapts to terminal size
- **Syntax highlighting**: Via chroma library

Example Model Structure:

```
type IntakeModel struct {
    // State
    step         IntakeStep         // Current step in wizard
    spec         *Specification     // Spec being created
    llmResponse  string             // LLM output
    err          error              // Error state

    // UI components
    textInput    textinput.Model    // Text input widget
    textArea     textarea.Model     // Multi-line editor
    progress     progress.Model     // Progress bar
    viewport     viewport.Model     // Scrollable area

    // Services
    llm          LLMService
    git          GitService
}

func (m IntakeModel) Init() tea.Cmd {
    return textinput.Blink
}

func (m IntakeModel) Update(msg tea.Msg) (tea.Model, tea.Cmd) {
    switch msg := msg.(type) {
    case tea.KeyMsg:
        return m.handleKeypress(msg)
    case llmResponseMsg:
        return m.handleLLMResponse(msg)
    case tea.WindowSizeMsg:
        return m.handleResize(msg)
    }
    return m, nil
}

func (m IntakeModel) View() string {
    return lipgloss.JoinVertical(
        lipgloss.Top,
        m.renderHeader(),
        m.renderContent(),
        m.renderFooter(),
    )
}
```

2. LLM Layer

Design: Provider-agnostic interface with pluggable backends

Architecture:

```
internal/llm/
├── interface.go           # LLM interface definition
├── ollama/
│   ├── client.go          # Ollama API client
│   └── streaming.go       # Streaming support
├── openai/
│   ├── client.go          # OpenAI API client
│   └── functions.go       # Function calling
├── anthropic/
│   └── client.go          # Claude API client
├── google/
│   └── client.go          # Gemini API client
├── azure/
│   └── client.go          # Azure OpenAI
├── custom/
│   └── client.go          # Custom endpoint
├── provider.go            # Provider factory
├── prompt_builder.go      # Prompt construction
└── response_parser.go     # Response parsing
```

Interface:

```go
type LLMService interface {
    // Generate text completion
    Generate(ctx context.Context, req GenerateRequest) (*GenerateResponse, error)

    // Stream text completion
    GenerateStream(ctx context.Context, req GenerateRequest) (<-chan GenerateChunk, er
ror)

    // List available models
    ListModels(ctx context.Context) ([]Model, error)

    // Get provider info
    Provider() string
}

type GenerateRequest struct {
    Prompt      string
    System      string              // System prompt
    Temperature float64
    MaxTokens   int
    Stop        []string
    Context     map[string]string   // Additional context
}

type GenerateResponse struct {
    Text         string
    TokensUsed   int
    FinishReason string
    Model        string
}
```

Provider Selection Logic:

```go
func NewLLMService(cfg *config.LLMConfig) (LLMService, error) {
    switch cfg.Provider {
    case "ollama":
        return ollama.NewClient(cfg.Endpoint, cfg.Model)
    case "openai":
        apiKey, _ := secrets.Get(cfg.APIKeyRef)
        return openai.NewClient(apiKey, cfg.Model)
    case "anthropic":
        apiKey, _ := secrets.Get(cfg.APIKeyRef)
        return anthropic.NewClient(apiKey, cfg.Model)
    // ... other providers
    default:
        return nil, fmt.Errorf("unknown provider: %s", cfg.Provider)
    }
}
```

Prompt Engineering:

```go
type PromptBuilder struct {
    mode     string
    context  *Context
    template *template.Template
}

func (pb *PromptBuilder) Build() string {
    // Load mode-specific template
    tmpl := pb.loadTemplate(pb.mode)

    // Inject context
    data := map[string]interface{}{
        "Mode":        pb.mode,
        "UserInput":   pb.context.UserInput,
        "CodeContext": pb.context.CodeSnippets,
        "Spec":        pb.context.ExistingSpec,
    }

    var buf bytes.Buffer
    tmpl.Execute(&buf, data)
    return buf.String()
}
```

3. GitHub Integration

Architecture:

```
internal/github/
├── oauth.go              # Device flow OAuth
├── app.go                # GitHub App API
├── client.go             # REST API client
├── operations/
│   ├── branch.go         # Branch operations
│   ├── commit.go         # Commit operations
│   ├── pr.go             # Pull request operations
│   └── issue.go          # Issue operations
├── webhook.go            # Webhook handling (future)
└── sync.go               # Bi-directional sync
```

Key Operations:

```go
type GitHubService interface {
    // OAuth
    InitiateDeviceFlow() (*DeviceCode, error)
    PollForToken(deviceCode string) (string, error)

    // Repository
    ListRepos() ([]Repository, error)
    GetRepo(owner, name string) (*Repository, error)

    // Branch & Commit
    CreateBranch(repo, branch, baseBranch string) error
    Commit(repo, branch string, files []FileChange) error

    // Pull Request
    CreatePR(repo string, pr *PullRequest) (*PullRequest, error)
    UpdatePR(repo string, prNumber int, updates *PRUpdate) error
    MergePR(repo string, prNumber int) error

    // Issues
    CreateIssue(repo string, issue *Issue) (*Issue, error)
}
```

PR Creation Workflow:

```go
func (s *GitHubService) CreateChangeOrderPR(
    repo string,
    spec *Specification,
    changes []CodeChange,
) (*PullRequest, error) {
    // 1. Create feature branch
    branchName := fmt.Sprintf("factory/change-%s", spec.ID)
    if err := s.CreateBranch(repo, branchName, "main"); err != nil {
        return nil, err
    }

    // 2. Commit changes
    files := make([]FileChange, len(changes))
    for i, change := range changes {
        files[i] = FileChange{
            Path:    change.Path,
            Content: change.NewContent,
        }
    }
    if err := s.Commit(repo, branchName, files); err != nil {
        return nil, err
    }

    // 3. Create PR
    pr := &PullRequest{
        Title:  spec.Title,
        Body:   s.generatePRBody(spec, changes),
        Head:   branchName,
        Base:   "main",
        Labels: []string{"code-factory", "automated"},
    }
    return s.CreatePR(repo, pr)
}
```

4. Mode Handlers

Each mode is a self-contained workflow with its own state machine.

Architecture:

```
internal/modes/
├── intake/
│   ├── handler.go          # Main handler
│   ├── wizard.go           # Step-by-step wizard
│   ├── spec_generator.go   # LLM-powered spec gen
│   └── validator.go        # Spec validation
├── review/
│   ├── handler.go
│   ├── analyzer.go         # Code analysis
│   ├── reporter.go         # Report generation
│   └── suggestions.go      # Improvement suggestions
├── change_order/
│   ├── handler.go
│   ├── planner.go          # Change planning
│   ├── implementer.go      # Code generation
│   └── pr_creator.go       # PR creation
└── rescue/
    ├── handler.go
    ├── detector.go         # Issue detection
    ├── debugger.go         # Problem diagnosis
    └── fixer.go            # Solution generator
```

Handler Interface:

```go
type ModeHandler interface {
    // Initialize mode
    Init(ctx context.Context) error

    // Run the mode workflow
    Run(ctx context.Context, input ModeInput) (*ModeOutput, error)

    // Get current state
    State() ModeState

    // Handle user input
    HandleInput(input string) error
}
```

Example: INTAKE Mode Flow:

```go
type IntakeHandler struct {
    llm    LLMService
    git    GitService
    github GitHubService
    state  IntakeState
}

func (h *IntakeHandler) Run(ctx context.Context, input ModeInput) (*ModeOutput,
error) {
    // Step 1: Gather requirements
    requirements, err := h.gatherRequirements(ctx, input)
    if err != nil {
        return nil, err
    }

    // Step 2: Generate spec via LLM
    spec, err := h.generateSpec(ctx, requirements)
    if err != nil {
        return nil, err
    }

    // Step 3: User review & edit
    spec, err = h.reviewSpecWithUser(ctx, spec)
    if err != nil {
        return nil, err
    }

    // Step 4: Save to /contracts/
    if err := h.saveSpec(spec); err != nil {
        return nil, err
    }

    // Step 5: Commit to git
    if err := h.git.Commit("Add spec: " + spec.Title); err != nil {
        return nil, err
    }

    // Step 6: Optional GitHub push
    if input.PushToGitHub {
        if err := h.github.PushSpec(spec); err != nil {
            return nil, err
        }
    }

    return &ModeOutput{
        Success: true,
        Spec:    spec,
        Message: "Specification created successfully",
    }, nil
}
```

5. Storage & File System

Storage Strategy: Git-native flat files

Directory Structure:

```
project-root/
├── contracts/                  # All specifications
│   ├── README.md               # Index & guide
│   ├── specs/
│   │   ├── feature-001.md      # Individual features
│   │   ├── feature-002.md
│   │   └── ...
│   ├── architecture/
│   │   ├── system-design.md    # High-level design
│   │   ├── data-model.md       # Data structures
│   │   └── api-spec.md         # API contracts
│   └── decisions/
│       ├── adr-001-use-go.md   # Architecture decisions
│       └── adr-002-tui.md
├── reports/                    # Generated reports
│   ├── review-2026-01-07.md    # Review outputs
│   ├── analysis-dashboard.md   # Aggregated metrics
│   └── change-log.md           # History of changes
├── .factory/                   # Factory metadata (gitignored)
│   ├── cache/                  # LLM response cache
│   └── temp/                   # Temporary files
└── .gitignore
```

File Operations:

```go
type FileService interface {
    // Read
    ReadSpec(path string) (*Specification, error)
    ListSpecs(dir string) ([]*Specification, error)

    // Write
    WriteSpec(spec *Specification) error
    UpdateSpec(spec *Specification) error
    DeleteSpec(path string) error

    // Reports
    WriteReport(report *Report) error

    // Templates
    LoadTemplate(name string) (*template.Template, error)
}
```

Spec Format:

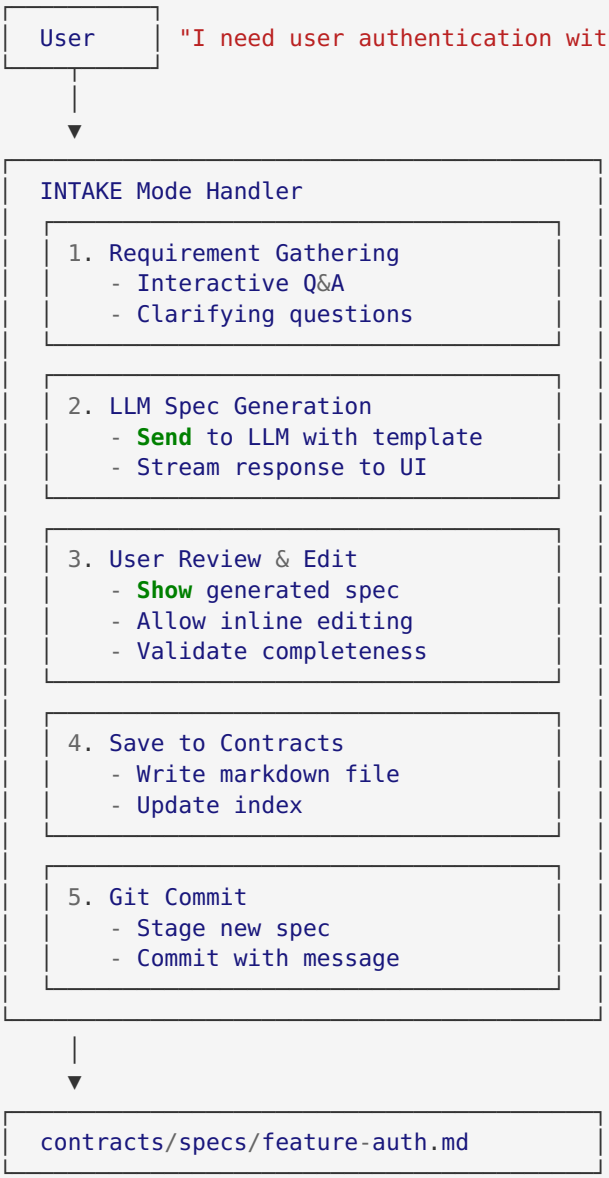
```
---
id: feature-001
title: User Authentication
status: draft
created: 2026-01-07
updated: 2026-01-07
author: johndoe
tags: [auth, security, backend]
---

# User Authentication

## Overview
This specification defines the user authentication system...

## Requirements
- Users must be able to sign up with email
- Passwords must be hashed with bcrypt
- JWT tokens for session management

## API Endpoints
- POST /api/auth/signup
- POST /api/auth/login
- POST /api/auth/logout

## Security Considerations
...
```

Git Operations:

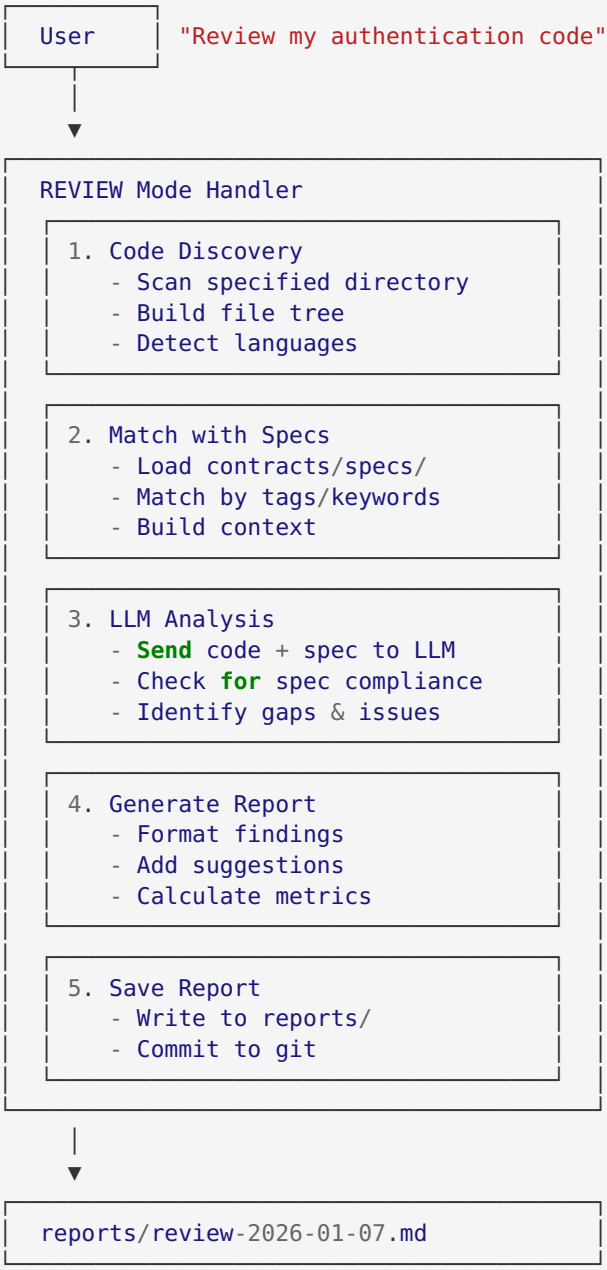
```go
type GitService interface {
    // Status
    Status() (*GitStatus, error)

    // Basic operations
    Add(files []string) error
    Commit(message string) error
    Push(remote, branch string) error
    Pull(remote, branch string) error

    // Branching
    CreateBranch(name string) error
    SwitchBranch(name string) error

    // History
    Log(limit int) ([]*Commit, error)
    Diff(ref1, ref2 string) (*Diff, error)
}
```

Data Flow

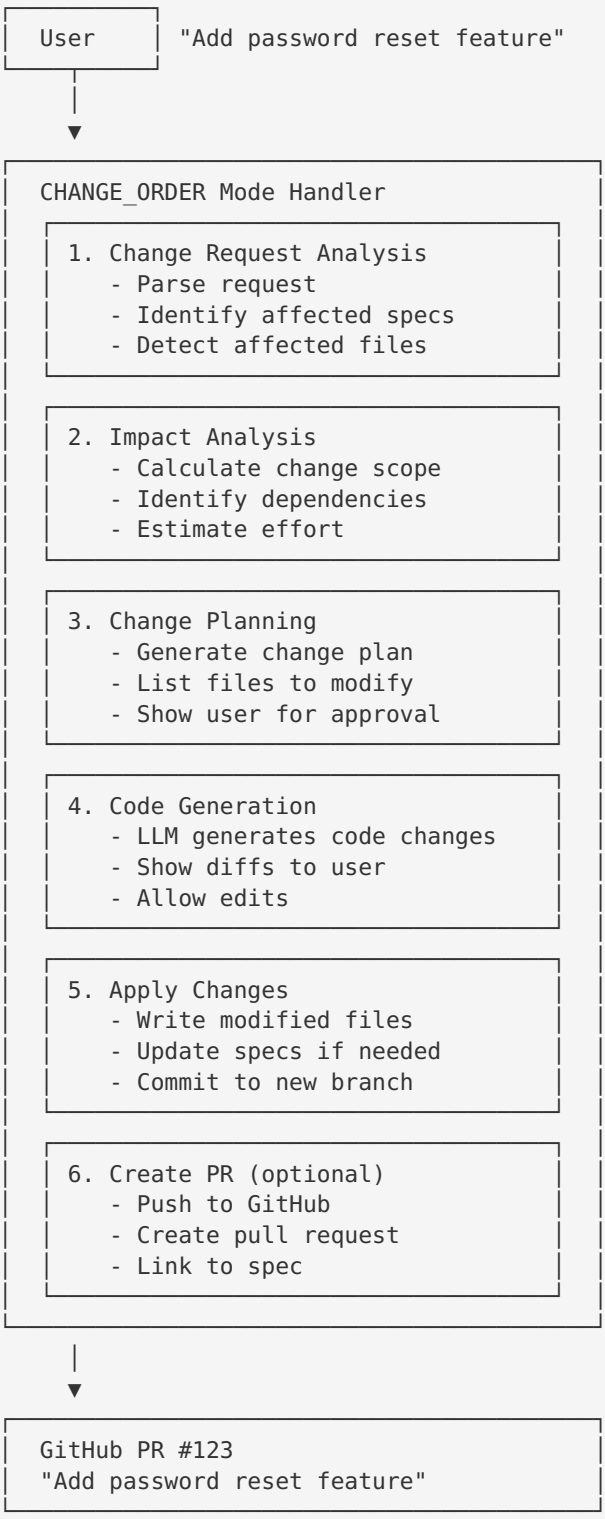
INTAKE Mode Flow



REVIEW Mode Flow



CHANGE_ORDER Mode Flow



Storage Model

Configuration Files

~/.config/factory/config.yaml:

```yaml
version: "1.0.0"
user:
  name: "John Doe"
  email: "john@example.com"

llm:
  provider: "ollama"
  endpoint: "http://localhost:11434"
  model: "llama3.2:latest"
  options:
    temperature: 0.7
    num_ctx: 8192

github:
  enabled: true
  username: "johndoe"
  token_ref: "github.oauth.token"
  app_installed: true

ui:
  theme: "auto"  # auto, light, dark
  animations: true
  syntax_highlight: true

projects:
  current: "/home/user/projects/my-app"
  recent:
    - "/home/user/projects/my-app"
    - "/home/user/projects/another-app"
```

Secrets (OS Keyring)

Stored securely, never in config files:

```
Service: "factory"
Accounts:
  - "llm.openai.key"           → "sk-..."
  - "llm.anthropic.key"        → "sk-ant-..."
  - "github.oauth.token"       → "ghp_..."
  - "github.installation.token" → "ghs_..."
```

Project Files

contracts/specs/template.md:

```
---
id: {auto-generated}
title: {Feature Title}
status: draft | approved | implemented | deprecated
created: {ISO date}
updated: {ISO date}
author: {username}
tags: [{tag1}, {tag2}]
priority: low | medium | high | critical
---

# {Feature Title}

## Overview
Brief description of the feature...

## Requirements

### Functional
- Requirement 1
- Requirement 2

### Non-Functional
- Performance targets
- Security requirements

## API Specification
Detailed API contracts...

## Data Model
Database schema, data structures...

## Implementation Notes
Technical considerations...

## Testing Strategy
How to verify implementation...

## Related Specs
- [Link to related spec](./other-spec.md)
```

Integration Points

External Services

1. **Ollama** (optional)
   - Endpoint: http://localhost:11434
   - Protocol: HTTP REST
   - Auth: None (local)

2. **OpenAI** (optional)
   - Endpoint: https://api.openai.com/v1
   - Protocol: HTTP REST
   - Auth: Bearer token

3. **Anthropic** (optional)
   - Endpoint: https://api.anthropic.com/v1
   - Protocol: HTTP REST
   - Auth: x-api-key header

4. **GitHub** (optional)
   - Endpoint: https://api.github.com
   - Protocol: HTTP REST
   - Auth: OAuth device flow

Internal Communication

TUI ↔ Core Services:
- Protocol: In-process function calls (Go)
- Pattern: Command pattern with async execution
- Error handling: Result types (value, error)

Web UI ↔ Core Services:
- Protocol: HTTP REST + WebSocket
- API: RESTful JSON
- Real-time updates: WebSocket for progress

Security Architecture

Threat Model

Assets:
- Source code
- API keys (OpenAI, GitHub)
- OAuth tokens
- Specifications (may contain sensitive info)

Threats:
1. API key leakage
2. Unauthorized access to GitHub repos
3. Malicious code injection via LLM
4. Credential theft from config files

Security Controls

1. **Secret Management**
   - OS keyring for sensitive data
   - No secrets in config files or logs
   - Encrypted fallback for systems without keyring

2. **Network Security**
   - HTTPS for all external APIs
   - Certificate validation
   - Timeout and retry limits

3. **Input Validation**
   - Sanitize all user inputs

   - Validate LLM outputs before execution
   - Path traversal prevention

4. **Audit Logging**
   - Log all LLM interactions
   - Log all GitHub API calls
   - Never log secrets

5. **Least Privilege**
   - GitHub OAuth: minimal scopes
   - File system: only project directory
   - Network: only necessary endpoints

Deployment Model

Distribution

Single Binary:

```
# macOS (Intel)
factory-darwin-amd64

# macOS (Apple Silicon)
factory-darwin-arm64

# Linux (x86_64)
factory-linux-amd64

# Linux (ARM64)
factory-linux-arm64

# Windows
factory-windows-amd64.exe
```

Installation:

```
# Via curl
curl -sSL https://factory.dev/install.sh | bash

# Via brew (macOS)
brew install code-factory

# Via apt (Ubuntu/Debian)
sudo apt install code-factory

# Manual
wget https://factory.dev/releases/v1.0.0/factory-linux-amd64
chmod +x factory-linux-amd64
sudo mv factory-linux-amd64 /usr/local/bin/factory
```

Configuration

Locations (in order of precedence):

1. Command-line flags: `--config=/path/to/config.yaml`

2. Environment variable: `FACTORY_CONFIG`
3. Project directory: `./.factory/config.yaml`
4. User config: `~/.config/factory/config.yaml`
5. System config: `/etc/factory/config.yaml`

Updates

Auto-update mechanism:

```
func CheckForUpdates() (*UpdateInfo, error) {
    resp, err := http.Get("https://api.github.com/repos/ssdajoker/Code-Factory/
releases/latest")
    // Parse response, compare versions
    // If newer version: prompt user to update
}

func UpdateSelf(version string) error {
    // Download new binary
    // Verify checksum
    // Replace current binary
    // Re-exec
}
```

User prompt:

```
New version available: v1.1.0 (current: v1.0.0)

Release notes:
  - Added support for Azure OpenAI
  - Improved spec parsing
  - Bug fixes

[Update now] [Skip this version] [Remind me later]
```

Scalability & Performance

Performance Targets

Metric	Target	Rationale
Startup time	< 100ms	Fast enough to feel instant
UI render	60 FPS	Smooth animations
LLM response (first token)	< 2s	Maintains flow
File operations	< 50ms	No perceived lag
Git operations	< 200ms	Acceptable for local ops

Optimization Strategies

1. **Lazy Loading**
   - Load specs on-demand, not at startup
   - Parse markdown only when needed
   - Cache parsed results

2. **Async Operations**
   - LLM calls in background
   - Git operations non-blocking
   - Progress feedback for long operations

3. **Response Streaming**
   - Stream LLM tokens to UI as received
   - Better perceived performance
   - User can read while waiting

4. **Caching**
   - Cache LLM responses (with TTL)
   - Cache parsed specs
   - Cache GitHub API responses

5. **Resource Limits**
   - Limit concurrent LLM requests (1)
   - Limit file size for analysis (10MB)
   - Limit number of files in review (100)

Scalability Considerations

Single User / Small Team:
- Current architecture is sufficient
- Local git + optional GitHub sync
- No server infrastructure needed

Future: Larger Teams
- Add centralized server for collaboration
- Real-time spec updates via WebSocket
- Shared LLM backend to reduce costs
- Role-based access control

Appendix

Technology Stack

Core:
- Language: Go 1.21+
- UI: Charm.sh (Bubble Tea, Lipgloss, Bubbles)
- Git: go-git
- Config: gopkg.in/yaml.v3

LLM Clients:
- Ollama: Direct HTTP client

- OpenAI: github.com/sashabaranov/go-openai
- Anthropic: Custom client
- Google: google.golang.org/genai

GitHub:
- github.com/google/go-github/v57
- golang.org/x/oauth2

Secrets:
- github.com/zalando/go-keyring

CLI:
- github.com/spf13/cobra
- github.com/spf13/viper

Future Enhancements

1. **Plugin System**
   - Executable hooks for custom workflows
   - Language-specific analyzers
   - Custom LLM providers

2. **Team Features**
   - Multi-user collaboration
   - Real-time co-editing of specs
   - Approval workflows

3. **CI/CD Integration**
   - GitHub Actions for automated reviews
   - Spec validation on PR
   - Automatic report generation

4. **Advanced AI**
   - Function calling for tool use
   - Multi-turn conversations
   - Context learning from project history

5. **IDE Integration**
   - VSCode extension
   - JetBrains plugin
   - Language server protocol

Revision History

Version	Date	Changes	Author
1.0.0	2026-01-07	Initial architecture	Code-Factory Team