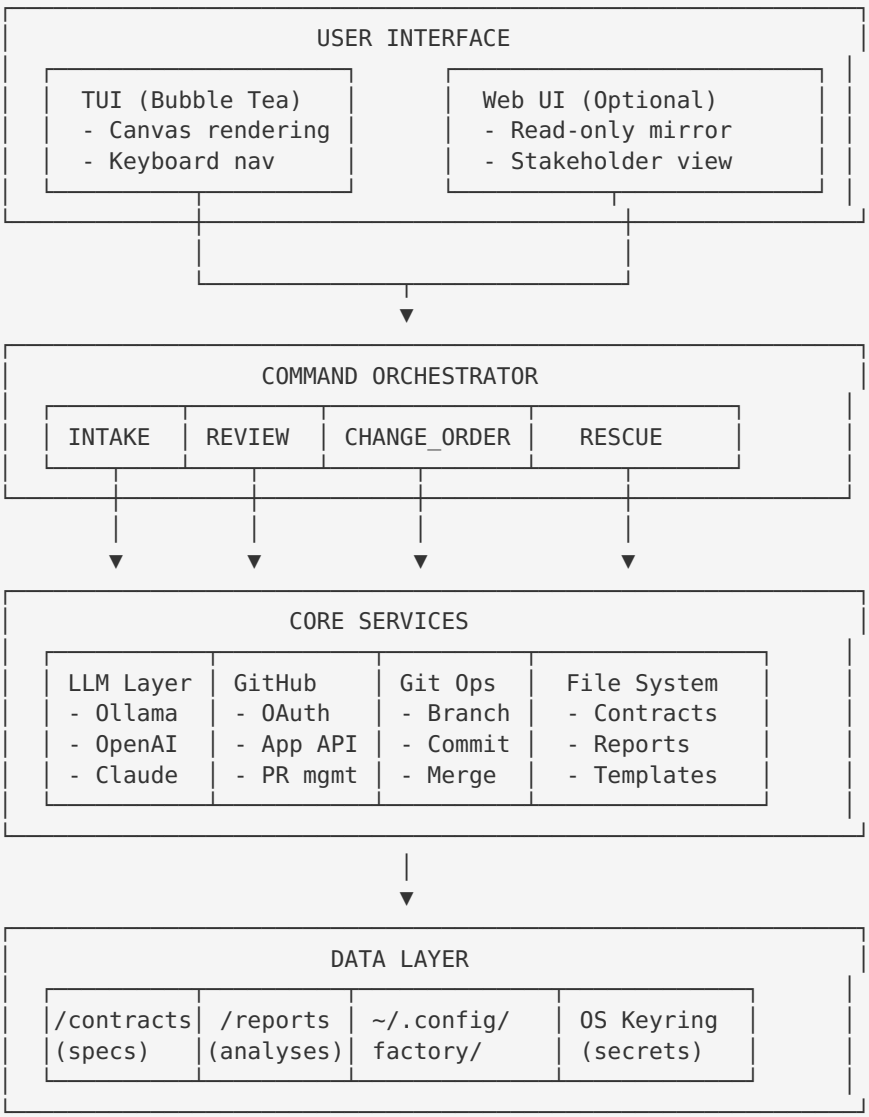
Code-Factory Technical Architecture

Version: 1.0.0
Last Updated: 2026-01-07

Overview

Code-Factory is a spec-driven software development system built as a single Go binary with zero dependencies. This document provides a technical overview of the system architecture, implementation patterns, and design decisions.

High-Level Architecture



Technology Stack

Core

- **Language:** Go 1.21+
- **Build:** Single static binary
- **Dependencies:** Zero runtime dependencies

User Interface

- **TUI:** Charm.sh ecosystem
- Bubble Tea (https://github.com/charmbracelet/bubbletea) - TUI framework
- Lipgloss (https://github.com/charmbracelet/lipgloss) - Styling
- Bubbles (https://github.com/charmbracelet/bubbles) - UI components
- **Web UI:** (Optional) Embedded HTTP server with htmx

LLM Integration

- **Ollama:** Direct HTTP client
- **OpenAI:** github.com/sashabaranov/go-openai
- **Anthropic:** Custom HTTP client
- **Google Gemini:** google.golang.org/genai

Git & GitHub

- **Git:** go-git/v5
- **GitHub:** github.com/google/go-github/v57
- **OAuth:** golang.org/x/oauth2

Utilities

- **CLI:** github.com/spf13/cobra
- **Config:** gopkg.in/yaml.v3
- **Secrets:** github.com/zalando/go-keyring
- **Markdown:** github.com/yuin/goldmark

Project Structure

```
Code-Factory/
├── cmd/
│   └── factory/
│       └── main.go              # Entry point
├── internal/
│   ├── tui/                     # Terminal UI
│   │   ├── app.go               # Main TUI app
│   │   ├── models/              # Bubble Tea models
│   │   │   ├── intake.go
│   │   │   ├── review.go
│   │   │   ├── change_order.go
│   │   │   └── rescue.go
│   │   ├── components/          # Reusable UI components
│   │   │   ├── header.go
│   │   │   ├── footer.go
│   │   │   ├── editor.go
│   │   │   ├── diff_viewer.go
│   │   │   └── progress.go
│   │   └── styles/              # UI styling
│   │       └── theme.go
│   ├── llm/                     # LLM integration
│   │   ├── interface.go         # LLM interface
│   │   ├── ollama/
│   │   │   └── client.go
│   │   ├── openai/
│   │   │   └── client.go
│   │   ├── anthropic/
│   │   │   └── client.go
│   │   └── provider.go          # Provider factory
│   ├── github/                  # GitHub integration
│   │   ├── oauth.go             # OAuth flow
│   │   ├── app.go               # App API
│   │   └── client.go            # REST client
│   ├── modes/                   # Mode handlers
│   │   ├── intake/
│   │   │   ├── handler.go
│   │   │   └── spec_generator.go
│   │   ├── review/
│   │   │   ├── handler.go
│   │   │   └── analyzer.go
│   │   ├── change_order/
│   │   │   ├── handler.go
│   │   │   └── implementer.go
│   │   └── rescue/
│   │       ├── handler.go
│   │       └── debugger.go
│   └── core/                    # Core utilities
│       ├── config/              # Configuration
│       ├── git/                 # Git operations
│       ├── files/               # File system
│       └── secrets/             # Secret management
├── contracts/                   # Specifications
│   ├── specs/
│   ├── architecture/
│   └── decisions/
├── reports/                     # Generated reports
├── docs/                        # Documentation
├── go.mod
├── go.sum
├── Makefile
└── README.md
```

Key Design Patterns

1. Bubble Tea (Elm Architecture)

All TUI components follow the Elm architecture:

```
type Model struct {
    // State
}

func (m Model) Init() tea.Cmd {
    // Initialize
}

func (m Model) Update(msg tea.Msg) (tea.Model, tea.Cmd) {
    // Handle events
}

func (m Model) View() string {
    // Render UI
}
```

2. Provider Pattern (LLM Abstraction)

```
type LLMService interface {
    Generate(ctx context.Context, req GenerateRequest) (*GenerateResponse, error)
    GenerateStream(ctx context.Context, req GenerateRequest) (<-chan GenerateChunk, er
ror)
    ListModels(ctx context.Context) ([]Model, error)
}

// Factory
func NewLLMService(provider string, config Config) (LLMService, error) {
    switch provider {
    case "ollama":
        return ollama.NewClient(config)
    case "openai":
        return openai.NewClient(config)
    // ...
    }
}
```

3. State Machine Pattern (Mode Handlers)

Each mode is a state machine:

```
type IntakeState int

const (
    StateInit IntakeState = iota
    StateGathering
    StateGenerating
    StateReview
    StateComplete
)

type IntakeHandler struct {
    state IntakeState
    // ...
}

func (h *IntakeHandler) Advance() error {
    switch h.state {
    case StateInit:
        return h.transitionToGathering()
    // ...
    }
}
```

4. Repository Pattern (Data Access)

```
type SpecRepository interface {
    List() ([]*Specification, error)
    Get(id string) (*Specification, error)
    Create(spec *Specification) error
    Update(spec *Specification) error
    Delete(id string) error
}

type FileSpecRepository struct {
    basePath string
}
```

Data Flow

Example: INTAKE Mode

```
User Input
    │
    ▼
IntakeModel (Bubble Tea)
    │
    ├─> IntakeHandler
    │       │
    │       ├─> LLM Service (generate spec)
    │       │       │
    │       │       └─> Ollama/OpenAI/Claude API
    │       │
    │       ├─> File Service (save spec)
    │       │       │
    │       │       └─> /contracts/specs/
    │       │
    │       └─> Git Service (commit)
    │               │
    │               └─> Local .git/
    │
    └─> GitHub Service (optional)
            │
            └─> GitHub API (create PR)
```

Build & Distribution

Build Process

```
# Build for current platform
go build -o factory ./cmd/factory

# Build for all platforms
make build-all

# Cross-compilation targets
GOOS=darwin GOARCH=amd64 go build -o factory-darwin-amd64 ./cmd/factory
GOOS=darwin GOARCH=arm64 go build -o factory-darwin-arm64 ./cmd/factory
GOOS=linux GOARCH=amd64 go build -o factory-linux-amd64 ./cmd/factory
GOOS=windows GOARCH=amd64 go build -o factory-windows-amd64.exe ./cmd/factory
```

Static Binary

```
# Build with CGO disabled for static binary
CGO_ENABLED=0 go build -ldflags="-s -w" -o factory ./cmd/factory

# Reduce binary size with upx (optional)
upx --best --lzma factory
```

Release Process

1. Tag version: `git tag v1.0.0`
2. Build binaries: `make build-all`
3. Create checksums: `shasum -a 256 factory-*`

4. Create GitHub release
5. Upload binaries
6. Update install scripts

Security Considerations

Secret Management

1. **Never store secrets in plain text**
   - Use OS keyring (preferred)
   - Fall back to encrypted file
   - Environment variables (CI/CD only)

2. **API keys**
   - Stored in keyring with service name “factory”
   - Referenced in config by key name
   - Never logged or exposed

3. **GitHub tokens**
   - OAuth tokens stored in keyring
   - Automatic refresh when expired
   - Proper scope limiting

Input Validation

1. **User inputs**
   - Sanitize all text inputs
   - Validate file paths (prevent traversal)
   - Limit input sizes

2. **LLM outputs**
   - Parse and validate before execution
   - Sandbox code execution
   - User approval for all changes

Network Security

1. **HTTPS only**
   - All external APIs use HTTPS
   - Certificate validation
   - TLS 1.2+ required

2. **Rate limiting**
   - Implement client-side rate limiting
   - Respect API rate limits
   - Exponential backoff on errors

Performance Optimization

Startup Time

- Lazy load configurations
- Minimal initialization
- Target: < 100ms startup

UI Responsiveness

- Async operations with progress indicators
- Stream LLM responses
- Non-blocking file I/O

Memory Usage

- Stream large files instead of loading entirely
- LRU cache for parsed specs
- Limit concurrent LLM requests

Binary Size

- Static compilation
- Strip debug symbols: `-ldflags="-s -w"`
- Optional: UPX compression
- Target: < 20MB uncompressed

Testing Strategy

Unit Tests

```
go test ./...
```

- Test all core logic
- Mock external dependencies
- Aim for 80%+ coverage

Integration Tests

```
go test -tags=integration ./...
```

- Test with real LLM providers
- Test GitHub integration
- Test file system operations

E2E Tests

```
make test-e2e
```

- Full workflow tests
- TUI interaction tests
- Real project scenarios

Monitoring & Observability

Logging

```
import "log/slog"

slog.Info("Operation completed", "mode", "intake", "duration", elapsed)
slog.Error("LLM request failed", "error", err, "provider", "openai")
```

Metrics

- Operation success/failure rates
- LLM response times
- Token usage
- File operation times

Audit Trail

- All operations logged to `.factory/audit.log`
- Include: timestamp, user, mode, action, outcome
- Rotate logs (max 10MB, keep 5 files)

Future Enhancements

Phase 2

- Plugin system for custom workflows
- Team collaboration features
- Real-time spec co-editing

Phase 3

- IDE extensions (VSCode, JetBrains)
- CI/CD integration
- Automated testing generation

Phase 4

- Multi-project management
- Analytics dashboard
- Knowledge graph of specifications

References

- Bubble Tea Documentation (https://github.com/charmbracelet/bubbletea)
- Go Project Layout (https://github.com/golang-standards/project-layout)
- Effective Go (https://go.dev/doc/effective_go)
- GitHub OAuth Device Flow (https://docs.github.com/en/apps/oauth-apps/building-oauth-apps/authorizing-oauth-apps#device-flow)

Document Status: Living Document
Maintained By: Code-Factory Core Team
Last Review: 2026-01-07