

# Cross-Cutting Engineering Principles

Document ID: TECH-000 Category: Cross-Cutting Last Updated: 2025-01-31 Status: Active

## PURPOSE

This document defines our **non-negotiable engineering principles** that apply across all technologies, platforms, and offerings. These principles reflect our corporate beliefs about how quality software should be built and represent the foundation upon which all technical decisions rest.

**All agents (Offerings Analyzer, Architect) MUST read this document before making technology or architecture recommendations.**

When SE Notes or client preferences conflict with these principles, agents MUST ask the Sales Engineer to clarify whether the client understands the trade-offs and explicitly wishes to deviate.

## CORE PRINCIPLES

### 1. TEST-DRIVEN DEVELOPMENT (TDD)

We are strong believers in **Test-Driven Development**. Tests are not an afterthought—they drive design.

Principle	Application
Tests First	Write tests before implementation code
Red-Green-Refactor	Follow the TDD cycle rigorously
AI-Assisted Development	Even when using AI tools, lead with tests first
Design Pressure	Use test difficulty as a signal for design problems

**For AI-assisted development:** When generating code, the AI should produce test cases first, then implementation. This applies to all agents and code generation scenarios.

**Non-Negotiable:** Every feature should have tests written before implementation begins.

## 2. DOMAIN-DRIVEN DESIGN (DDD)

We embrace **Domain-Driven Design** as our approach to modeling complex business domains.

Concept	Our Application
<b>Ubiquitous Language</b>	Domain terms from the glossary permeate code, tests, and documentation
<b>Bounded Contexts</b>	Clear boundaries between subdomains with explicit contracts
<b>Aggregates</b>	Design around consistency boundaries, not data structures
<b>Domain Events</b>	Use events to communicate between bounded contexts
<b>Anti-Corruption Layers</b>	Protect domain model from external system pollution

**Integration with Pipeline:** The Analyst Brief's client glossary and domain model should directly inform entity names, service boundaries, and API contracts in the architecture.

## 3. SHIFT-LEFT

We practice **Shift-Left** across all quality dimensions—finding issues earlier is always cheaper than finding them later.

Area	Shift-Left Practice
<b>Testing</b>	Unit tests run on every save; integration tests on every commit
<b>Security</b>	SAST/DAST in CI pipeline; threat modeling in design phase
<b>Performance</b>	Performance budgets defined upfront; load tests in CI
<b>Accessibility</b>	Automated a11y checks in development; manual audits pre-release
<b>Documentation</b>	API docs generated from code; architecture decisions documented when made

**Quality Gates:** No code merges without passing fast tests. No deployment without passing slow tests.

## 4. CONTAINERIZATION

We believe in **containerization** as the standard deployment unit for all services.

Principle	Application
<b>Container-First</b>	Design for containers from the start, not as an afterthought
<b>Immutable Infrastructure</b>	Containers are built once, promoted through environments
<b>12-Factor App</b>	Follow 12-factor principles for container-native applications
<b>Local Parity</b>	Development containers match production configuration
<b>Orchestration Ready</b>	Design for Kubernetes/container orchestration from day one

**Exceptions:** Only client-side applications (mobile, desktop, browser) are exempt. Backend services, APIs, workers, and scheduled jobs should all be containerized.

## 5. LOCAL DISCONNECTED DEVELOPMENT

Developers should be able to **work productively without network connectivity**.

Requirement	Implementation
<b>Offline-Capable Build</b>	All dependencies cached locally; no build-time network calls
<b>Local Data</b>	Seed data and mocks available for all external dependencies
<b>Containerized Dependencies</b>	Databases, queues, caches run locally via Docker Compose
<b>Service Virtualization</b>	External APIs stubbed for local development
<b>Fast Feedback</b>	Full fast test suite runs locally in under 5 minutes

**Goal:** A developer should be able to clone the repo, run a single command, and have a fully functional development environment—even on an airplane.

## 6. OBSERVABILITY

All systems must be **observable**—we cannot manage what we cannot measure.

Principle	Application
<b>OpenTelemetry</b>	Use OpenTelemetry as the standard for traces, metrics, and logs
<b>Health Endpoints</b>	Every service exposes <code>/health</code> for basic liveness
<b>Liveness Probes</b>	Kubernetes-ready probes indicating the service is running
<b>Readiness Probes</b>	Indicate when service is ready to accept traffic
<b>Distributed Tracing</b>	Trace requests across service boundaries with correlation IDs
<b>Metrics Export</b>	Export metrics in Prometheus format or via OTLP

#### Health Endpoint Requirements:

Endpoint	Purpose	Response
<code>/health</code> or <code>/healthz</code>	Basic liveness	200 OK if process is running
<code>/health/ready</code>	Readiness check	200 OK if dependencies are available
<code>/health/live</code>	Liveness check	200 OK if process is not deadlocked

**Non-Negotiable:** Even non-containerized applications **MUST** expose health endpoints. This enables monitoring, load balancer integration, and operational visibility regardless of deployment model.

#### OpenTelemetry Guidance:

- Instrument with OTEL SDK from day one, not as an afterthought
- Propagate trace context across all service boundaries
- Use semantic conventions for span and metric names
- Export to a collector, not directly to backends (flexibility to change backends)

## 7. STRUCTURED LOGGING

All application logging must be **structured** (machine-readable) rather than plain text.

Principle	Application
<b>JSON Format</b>	Logs emitted as JSON objects, not free-form text
<b>Consistent Fields</b>	Standard fields: timestamp, level, message, correlationId, service
<b>Correlation</b>	Include trace/span IDs to correlate logs with traces
<b>No Sensitive Data</b>	Never log PII, credentials, or secrets
<b>Contextual Enrichment</b>	Include relevant context (userId, orderId) without over-logging

### Standard Log Fields:

Field	Required	Description
timestamp	Yes	ISO 8601 format with timezone
level	Yes	trace, debug, info, warn, error, fatal
message	Yes	Human-readable description
service	Yes	Service/application name
correlationId	Yes	Request correlation ID
traceId	When tracing	OpenTelemetry trace ID
spanId	When tracing	OpenTelemetry span ID

### Log Level Guidance:

Level	When to Use
error	Failures requiring attention; include stack traces
warn	Recoverable issues, degraded operation
info	Business events, request lifecycle
debug	Development diagnostics; disabled in production
trace	Verbose debugging; never in production

**Non-Negotiable:** Plain-text `Console.WriteLine` or `print()` statements are not acceptable for production logging.

## 8. OPENAPI

All HTTP APIs must be **documented with OpenAPI** (formerly Swagger).

Principle	Application
<b>API-First Option</b>	Consider designing OpenAPI spec first, then implementing
<b>Auto-Generation</b>	Generate OpenAPI from code annotations/decorators
<b>Developer Experience</b>	Provide interactive documentation UI for API exploration
<b>Versioning</b>	Include API version in spec; document breaking changes
<b>Examples</b>	Include request/response examples in spec

### OpenAPI Requirements:

Aspect	Requirement
<b>Spec Generation</b>	Automated from code, not manually maintained
<b>Documentation UI</b>	Interactive docs accessible in non-production environments
<b>Schemas</b>	All request/response bodies have typed schemas
<b>Authentication</b>	Security schemes documented in spec
<b>Errors</b>	Error responses documented with examples

### Developer Experience Tools:

Approach	Recommendation
<b>.NET</b>	Scalar (preferred over Swagger UI)
<b>Java/Spring</b>	SpringDoc OpenAPI with Swagger UI
<b>Node.js</b>	Swagger UI Express or Scalar
<b>Frontend Integration</b>	Generate TypeScript clients from OpenAPI spec

**Non-Negotiable:** Every HTTP API endpoint must appear in the OpenAPI specification with documented request/response schemas.

## 9. SOLID AND SEPARATION OF CONCERNS

We design systems following **SOLID principles** and clear **separation of concerns**.

Principle	Definition	Application
<b>S - Single Responsibility</b>	A class should have one reason to change	One purpose per class; split when responsibilities diverge
<b>O - Open/Closed</b>	Open for extension, closed for modification	Use interfaces/abstractions; extend via composition
<b>L - Liskov Substitution</b>	Subtypes must be substitutable for base types	Derived classes honor base contracts; avoid surprises
<b>I - Interface Segregation</b>	Clients shouldn't depend on unused methods	Small, focused interfaces; split fat interfaces
<b>D - Dependency Inversion</b>	Depend on abstractions, not concretions	Inject dependencies; high-level modules own interfaces

#### Separation of Concerns in Practice:

Layer	Responsibility	Depends On
<b>Presentation</b>	UI rendering, input handling	Application
<b>Application</b>	Use cases, orchestration, DTOs	Domain
<b>Domain</b>	Business logic, entities, rules	Nothing (pure)
<b>Infrastructure</b>	Data access, external services, I/O	Domain (implements interfaces)

#### Guidance:

- Domain layer has **zero dependencies** on infrastructure or frameworks
- Use Dependency Injection to wire layers together
- Avoid "smart" UI components that contain business logic
- Test business rules in isolation via the domain layer
- Infrastructure implements domain-defined interfaces (Repository pattern)

**Non-Negotiable:** Business logic must not live in controllers, UI components, or infrastructure code. The domain layer is the authoritative source for business rules.

## 10. OPEN SOURCE LICENSE COMPLIANCE

We have **zero tolerance for viral/copyleft licenses** in our deliverables.

License Category	Examples	Policy
<b>Permissive (Allowed)</b>	MIT, Apache 2.0, BSD, ISC, Unlicense	✅ Approved for all use
<b>Weak Copyleft (Review Required)</b>	LGPL, MPL, EPL	⚠️ Requires legal review; generally OK for dynamic linking
<b>Strong Copyleft (Prohibited)</b>	GPL, AGPL, SSPL, CPAL	❌ <b>NEVER</b> allowed in our deliverables
<b>Unclear/Custom</b>	WTFPL, proprietary, no license	⚠️ Requires manual review

### CI Pipeline Enforcement:

All projects **MUST** include automated license scanning in CI that **fails the build** on violations:

Aspect	Requirement
<b>Scanning Scope</b>	Both production AND development dependencies
<b>Build Failure</b>	CI fails if prohibited license detected
<b>Configuration</b>	Project-level config file to customize allowed/blocked lists
<b>Transitive Dependencies</b>	Scan full dependency tree, not just direct dependencies
<b>Audit Trail</b>	License scan results stored as build artifacts

### Project Configuration:

Every project should include a license policy file that can be tightened beyond the defaults:

```
# Example: .license-checker.json, license-allowed.txt, or similar
{
  "allowed": ["MIT", "Apache-2.0", "BSD-2-Clause", "BSD-3-Clause", "ISC"],
  "blocked": ["GPL-*", "AGPL-*", "SSPL-*", "CPAL-*"],
  "review_required": ["LGPL-*", "MPL-*", "EPL-*"],
  "fail_on_unknown": true
}
```

### Why This Matters:

- Viral licenses can legally "infect" client codebases
- AGPL applies even to network use (SaaS/APIs)
- Client legal teams require clean license audits
- Open source compliance is a contractual obligation

**Non-Negotiable:** Every build must pass license compliance scanning. Projects with GPL/AGPL dependencies will not be delivered.

## 11. SOFTWARE BILL OF MATERIALS (SBOM)

Every build must produce a **Software Bill of Materials** as a build artifact.

Aspect	Requirement
<b>Format</b>	CycloneDX (preferred) or SPDX
<b>Generation</b>	Automated as part of every build
<b>Scope</b>	Both production AND development dependencies
<b>Transitive Inclusion</b>	Full dependency tree, not just direct dependencies
<b>Artifact Storage</b>	SBOM file stored alongside build artifacts
<b>Versioning</b>	SBOM includes version/hash of each component

### Why SBOM Matters:

Use Case	Benefit
<b>Supply Chain Security</b>	Identify vulnerable dependencies (Log4Shell, etc.)
<b>License Compliance</b>	Cross-reference with license scanning
<b>Incident Response</b>	Quickly identify affected systems when CVE announced
<b>Audit Requirements</b>	Regulatory compliance (Executive Order 14028, etc.)
<b>Client Handoff</b>	Transparent dependency disclosure

### SBOM Content Requirements:

Field	Required	Description
Component Name	Yes	Package/library name
Version	Yes	Exact version used
Package URL (PURL)	Yes	Canonical identifier
License	Yes	SPDX license identifier
Hashes	Recommended	SHA-256 of package
Supplier	Recommended	Package maintainer/vendor
Dependencies	Yes	Transitive dependency tree

### Production vs Development:

Dependency Type	Include in SBOM	Rationale
Runtime (prod)	Yes	Deployed to production
Build-time (dev)	Yes	Affects build integrity
Test-only	Yes	Can introduce vulnerabilities in CI
Optional/Peer	Yes, if resolved	May be activated at runtime

**Non-Negotiable:** Every release build must include a CycloneDX or SPDX SBOM. Builds without SBOM generation are not deployable.

## TESTING PHILOSOPHY

### THE TWO CATEGORIES OF TESTS

While we recognize the traditional test pyramid (Unit, Integration, Smoke, E2E), we believe all tests fundamentally fall into **two categories**:

Category	Characteristics	Target
<b>Fast Tests</b>	Quick execution (<100ms each), non-fragile, no external dependencies, run locally	80%+ of test suite
<b>Slow Tests</b>	Involve more of the stack, may require setup/teardown, higher fragility risk	20% or less of test suite

### FAST TESTS

Fast tests are the **backbone of developer productivity**.

Attribute	Requirement
<b>Execution Time</b>	Individual test < 100ms; full suite < 5 minutes
<b>Isolation</b>	No shared state, no external dependencies
<b>Determinism</b>	Same inputs always produce same outputs
<b>Locality</b>	Run entirely on developer's machine
<b>Parallelization</b>	Tests can run concurrently without interference

#### What counts as Fast:

- Unit tests (pure logic, no I/O)
- Component tests with mocked dependencies
- Contract tests with stubbed collaborators

In-memory integration tests (e.g., in-memory database)

## SLOW TESTS

Slow tests validate **system behavior at integration boundaries** but come with trade-offs.

Attribute	Reality
Execution Time	Seconds to minutes per test
Dependencies	Require real databases, services, or infrastructure
Reset Complexity	May need data cleanup, container restarts
Fragility Risk	Network issues, timing, external service changes
Environment	Typically run in CI, not on every save

### What counts as Slow:

- Integration tests against real databases
- API tests against deployed services
- End-to-end tests through UI
- Smoke tests against staging environments
- Performance/load tests

## TEST RATIO GUIDANCE

Scenario	Fast:Slow Ratio	Rationale
Typical Service	80:20	Most logic testable in isolation
Integration-Heavy	70:30	More boundary validation needed
UI-Heavy Application	75:25	Component tests replace many E2E tests
Data Pipeline	60:40	More integration validation inherently needed

**Principle:** When you find yourself writing slow tests, ask: "Can I redesign to make this testable with a fast test instead?"

# DESIGN PATTERNS

## GANG OF FOUR PATTERNS

We draw heavily from the **Gang of Four (GoF) design patterns**. These are the vocabulary of object-oriented design.

Pattern Category	Common Applications
<b>Creational</b>	Factory, Builder, Singleton (sparingly)
<b>Structural</b>	Adapter, Decorator, Facade, Composite
<b>Behavioral</b>	Strategy, Observer, Command, State

### Guidance:

- Prefer composition over inheritance
- Use patterns to solve actual problems, not speculatively
- Name classes/methods to reveal pattern usage (e.g., `OrderFactory` , `PricingStrategy` )
- Document pattern usage in code comments when non-obvious

## ENTERPRISE INTEGRATION PATTERNS

For system integration, we follow **Enterprise Integration Patterns** by Gregor Hohpe and Bobby Woolf.

**Reference:** [enterpriseintegrationpatterns.com](http://enterpriseintegrationpatterns.com)

Pattern Category	Key Patterns We Use
<b>Messaging Channels</b>	Point-to-Point, Publish-Subscribe, Dead Letter
<b>Message Construction</b>	Command Message, Event Message, Document Message
<b>Message Routing</b>	Content-Based Router, Message Filter, Splitter, Aggregator
<b>Message Transformation</b>	Envelope Wrapper, Content Enricher, Normalizer
<b>Endpoints</b>	Polling Consumer, Event-Driven Consumer, Competing Consumers

### Guidance:

- Use EIP vocabulary when designing integrations
- Prefer asynchronous messaging over synchronous calls for cross-service communication
- Design for idempotency—messages may be delivered more than once
- Implement the Outbox Pattern for reliable event publishing

## APPLICATION TO OFFERINGS

Each offering should align with these cross-cutting principles:

Offering	Key Principle Applications
<b>Application Modernization</b>	Containerize legacy; add tests before refactoring; add observability; generate SBOM for existing dependencies
<b>Custom Software Development</b>	TDD from day one; DDD for domain modeling; OpenAPI for APIs; SOLID from the start; license scanning in CI
<b>Enterprise Platform Engineering</b>	EIP for integrations; shift-left for quality; structured logging; SBOM for supply chain visibility
<b>Cloud Native Development</b>	Container-first; 12-factor adherence; OpenTelemetry tracing; clean dependency licenses
<b>DevOps &amp; Automation</b>	Local dev parity; fast test pipelines; health endpoints; license + SBOM gates in pipelines
<b>AI Solution Development</b>	Test AI components; containerize inference services; metrics export; audit ML library licenses
<b>Intelligent Applications</b>	DDD for domain logic; TDD for non-AI components; structured logging; SOLID for extensibility
<b>Modern Data Engineering</b>	Test data transformations; containerize pipelines; observability; SBOM for data tool dependencies

## CONFLICT RESOLUTION

When client preferences, SE Notes, or discovery materials suggest approaches that conflict with these principles:

### FOR AGENTS (OFFERINGS ANALYZER, ARCHITECT)

- Identify the conflict** explicitly in your analysis
- Do NOT silently override** these principles
- Ask the Sales Engineer** to clarify with the client:

#### CROSS-CUTTING PRINCIPLE CONFLICT

Principle: [Name of principle]  
 Our Standard: [What we normally do]  
 Client Preference: [What was requested] [Citation]

This represents a deviation from our core engineering principles.

Questions for clarification:

1. Is the client aware of the trade-offs involved?
2. Is there a specific constraint driving this preference?
3. Should we propose our standard approach as an alternative?

Please clarify how to proceed.

### ACCEPTABLE DEVIATIONS

Some situations may warrant deviation:

Situation	Potential Deviation	Required Documentation
<b>Legacy Integration</b>	Non-containerized component	Document modernization path
<b>Client Mandate</b>	Specific technology/approach	Document trade-offs accepted
<b>Regulatory Requirement</b>	Non-standard testing approach	Document compliance rationale
<b>Timeline Constraint</b>	Reduced test coverage initially	Document technical debt and remediation plan

#### All deviations must be:

- Explicitly documented in the Architecture Brief
- Acknowledged by the Sales Engineer
- Accompanied by a remediation plan when applicable

## CITATION FORMAT

When referencing this document:

**Full document:** [TECH:cross-cutting]

**Specific section:** [TECH:cross-cutting:tdd], [TECH:cross-cutting:testing-philosophy]

**Observability sections:** [TECH:cross-cutting:observability], [TECH:cross-cutting:structured-logging], [TECH:cross-cutting:openapi]

**Quality sections:** [TECH:cross-cutting:solid] , [TECH:cross-cutting:license-compliance] , [TECH:cross-cutting:sbom]

## CHANGE HISTORY

Date	Author	Reviewer	Description
2025-01-31	Claude	Tim Rayburn	Initial creation - TDD, DDD, Shift-Left, Containerization, Local Dev, Testing Philosophy (Fast/Slow), GoF & EIP patterns
2026-01-31	Claude	Tim Rayburn	Add Observability (OpenTelemetry, Health/Liveness), Structured Logging, and OpenAPI sections
2026-01-31	Claude	Tim Rayburn	Add SOLID principles, Open Source License Compliance (no viral licenses, CI enforcement), and SBOM generation requirements