QUALITATIVE_SUCCESS_CRITERIA

1 Djed Qualitative Success Criteria

Measuring success through human experience, not just metrics

1.1 🎯 Philosophy

Quantitative metrics (downloads, test coverage, bundle size) tell us *what* happened. Qualitative criteria tell us *why it matters* and *how it feels* to use Djed.

Core Questions: - Does this make developers' lives better? - Would we be proud to recommend this to others? - Does this align with our vision of infrastructure excellence?

1.2 🌟 Overall Djed Vision Success Criteria

1.2.1 The "5-Minute Test"

Scenario: A developer new to LUXOR joins the team

Success Looks Like: - ✅ They can read the Djed README and understand what it offers in < 2 minutes - ✅ They can add @djed/logger to their project and see logs in < 5 minutes - ✅ They feel **confident** the package is production-ready (tests, docs, examples) - ✅ They say "This is exactly what I needed" not "I guess this works"

Failure Looks Like: - ❌ They're confused about what Djed is for - ❌ They copy-paste code without understanding it - ❌ They abandon it for a simpler alternative - ❌ They ask "Is this safe to use in production?"

1.2.2 The "Production Confidence Test"

Scenario: A senior engineer reviews Djed for production use

Success Looks Like: - ✅ They audit the code and find it **clean, well-tested, and maintainable** - ✅ They check the documentation and find **answers to all their questions** - ✅ They review the examples and see **real-world patterns**, not toy demos - ✅ They approve it with "This is professional-grade infrastructure"

Failure Looks Like: - ❌ They find untested edge cases or security concerns - ❌ They can't understand how to configure it for production - ❌ They say "This feels like a side project, not infrastructure" - ❌ They block usage until it's "more mature"

1.2.3 The "Ecosystem Coherence Test"

Scenario: A developer uses multiple Djed packages together

Success Looks Like: - ✅ Packages **feel like they belong together** (consistent APIs, naming, patterns) - ✅ Integration is **obvious and natural** (logger + config + errors work seamlessly) - ✅ Documentation shows **how packages compose**, not just individual usage - ✅ They think "Whoever designed this thought about the whole system"

Failure Looks Like: - ❌ Each package feels like it was built by different teams - ❌ Integration requires hacks or workarounds - ❌ No guidance on how packages work together - ❌ They say "Why isn't this one library instead of many packages?"

1.3 📦 Phase 1: @djed/logger Success Criteria

1.3.1 Package Quality

1.3.1.1 L1 API (Novice): "It Just Works"

Scenario: Junior developer needs logging in their first Node.js app

Success Looks Like: - ✅ They run `const logger = createLogger();` and it **works immediately** - ✅ They see **formatted, timestamped logs** in the console without configuration - ✅ They feel **empowered**, not overwhelmed by options - ✅ They think "This is simpler than console.log but way better"

Failure Looks Like: - ❌ They get configuration errors or cryptic warnings - ❌ Output is ugly or unreadable - ❌ They give up and use `console.log` instead - ❌ They think "This is too complicated for logging"

1.3.1.2 L2 API (Intermediate): "Control When I Need It"

Scenario: Mid-level developer needs to customize logging for different environments

Success Looks Like: - ✅ They find the configuration options **intuitive and predictable** - ✅ They can add file logging **without reading docs** (autocomplete + types guide them) - ✅ They configure different log levels for dev/prod **with confidence** - ✅ They think "I have control, but it doesn't overwhelm me"

Failure Looks Like: - ❌ Configuration options are confusing or poorly named - ❌ TypeScript types don't help them understand what's possible - ❌ They have to read docs for every small change - ❌ They think "Why is this so hard to customize?"

1.3.1.3 L3 API (Expert): "Power for Edge Cases"

Scenario: Senior engineer needs custom transports and formats

Success Looks Like: - ✅ They can **drop down to Winston** for advanced features without friction - ✅ Custom transports and formats **work as expected** - ✅ Documentation shows **realistic advanced examples**, not just "it's possible" - ✅ They think "Good abstractions that don't get in my way"

Failure Looks Like: - ❌ Abstractions make advanced usage impossible or hacky - ❌ No documentation on how to extend - ❌ They fork the package to add features - ❌ They think "I should have just used Winston directly"

1.3.2 Documentation Quality

1.3.2.1 README: "First Impressions Matter"

Scenario: Developer discovers @djed/logger on npm or GitHub

Success Looks Like: - ✅ They understand **what it does in 10 seconds** (clear description + examples) - ✅ They see **quality signals** (badges, tests, bundle size) that build trust - ✅ They find the **Quick-Start link** and click it immediately - ✅ They think "This looks professional and well-maintained"

Failure Looks Like: - ❌ README is vague about what the package actually does - ❌ No clear entry point for getting started - ❌ Looks abandoned or incomplete - ❌ They think "I'll find something else"

1.3.2.2 Quick-Start Guide: "Speed to Success"

Scenario: Developer wants to evaluate @djed/logger quickly

Success Looks Like: - ✅ They get **working code in under 5 minutes** by copy-pasting examples - ✅ Examples are **realistic** (not just "hello world") - ✅ They learn **best practices** naturally through examples - ✅ They think "I can see exactly how this works in my project"

Failure Looks Like: - ❌ Examples are too simplistic to be useful - ❌ Guide assumes knowledge they don't have - ❌ No guidance on production setup - ❌ They think "This doesn't answer my real questions"

1.3.2.3 API Documentation: "Reference When Needed"

Scenario: Developer needs to look up a specific option or method

Success Looks Like: - ✅ They find the information **quickly** (good structure, search, TOC) - ✅ Every option is **explained with examples**, not just type signatures - ✅ Edge cases and gotchas are **documented proactively** - ✅ They think "This documentation respects my time"

Failure Looks Like: - ❌ Documentation is hard to navigate - ❌ Options are listed but not explained - ❌ They have to read source code to understand behavior - ❌ They think "This feels incomplete"

1.3.3 Testing & Quality

1.3.3.1 Test Coverage: "Confidence to Deploy"

Scenario: Developer reviews test suite before using in production

Success Looks Like: - ✅ Tests cover **real-world scenarios**, not just happy paths - ✅ Test names **explain behavior** clearly (readable as documentation) - ✅ Edge cases and error conditions are **thoroughly tested** - ✅ They think "These developers care about correctness"

Failure Looks Like: - ❌ Tests are trivial or redundant - ❌ Critical paths are untested - ❌ Test names are cryptic (test1, test2) - ❌ They think "This is just for the coverage numbers"

1.3.3.2 Bundle Size: "Respect for User's App"

Scenario: Developer adds @djed/logger to size-sensitive application

Success Looks Like: - ✅ Bundle impact is **negligible** (< 2 KB) - ✅ No surprise dependencies in their bundle analyzer - ✅ Tree-shaking works as expected - ✅ They think "This won't bloat my application"

Failure Looks Like: - ❌ Package adds unexpected weight - ❌ Dependencies bring in bloat - ❌ Can't tree-shake unused code - ❌ They think "This is too heavy for what it does"

1.4 🚀 Phase 2A: Core Infrastructure Packages

1.4.1 @djed/config Success Criteria

1.4.1.1 The "No More .env Bugs" Test

Scenario: Developer uses @djed/config instead of manual env var parsing

Success Looks Like: - ✅ Missing required env vars are **caught at startup**, not in production - ✅ Type errors (string vs number) are **prevented by schema validation** - ✅ They get **clear error messages** pointing to the exact problem - ✅ They think "This saves me from stupid mistakes"

Failure Looks Like: - ❌ Errors are cryptic or unhelpful - ❌ Validation happens too late (after app starts) - ❌ No guidance on how to fix issues - ❌ They think "This adds complexity without value"

1.4.1.2 The "Environment Parity" Test

Scenario: Developer configures different settings for dev/staging/prod

Success Looks Like: - ✅ Configuration **hierarchy is obvious** (.env.local overrides .env) - ✅ They can **preview what config will load** before running the app - ✅ Sensitive values (secrets) are **clearly marked and protected** - ✅ They think "I trust this configuration won't leak secrets"

Failure Looks Like: - ❌ Override behavior is surprising or undocumented - ❌ No way to validate config without running the app - ❌ Secrets accidentally logged or exposed - ❌ They think "I'm not sure what values are actually being used"

1.4.1.3 The "Type Safety Joy" Test

Scenario: Developer uses TypeScript with @djed/config

Success Looks Like: - ✅ Their editor **autocompletes config keys** perfectly - ✅ Type errors are **caught at compile time**, not runtime - ✅ Refactoring config is **safe** (renames cascade automatically) - ✅ They think "This is how config should work in TypeScript"

Failure Looks Like: - ❌ Types are `any` or too loose - ❌ Autocomplete doesn't work - ❌ Type errors slip through to runtime - ❌ They think "I'll just use process.env"

1.4.2 @djed/errors Success Criteria

1.4.2.1 The "Debug Faster" Test

Scenario: Developer investigates a production error

Success Looks Like: - ✅ Error message includes **all context** needed to diagnose (user ID, request ID, etc.) - ✅ Stack traces are **clean and readable** (no noise from library internals) - ✅ Error codes/types make it **easy to search logs** and find related issues - ✅ They think "I know exactly what went wrong and where"

Failure Looks Like: - ❌ Generic error messages like "Something went wrong" - ❌ Missing context (what user? what request?) - ❌ Stack traces are cluttered and unhelpful - ❌ They think "I have no idea what caused this"

1.4.2.2 The "Consistent API Responses" Test

Scenario: Frontend developer integrates with API using @djed/errors

Success Looks Like: - ✅ All errors have **consistent structure** (code, message, context) - ✅ HTTP status codes **match semantic meaning** (404 for NotFound, 400 for Validation) - ✅ Error messages are **user-friendly when needed**, technical when debugging - ✅ They think "Error handling is predictable across all endpoints"

Failure Looks Like: - ❌ Error format varies by endpoint - ❌ Status codes don't match error types - ❌ Error messages expose internal details to users - ❌ They think "I need custom handling for every error type"

1.4.2.3 The "Monitoring Integration" Test

Scenario: DevOps engineer integrates errors with monitoring (Sentry, Datadog)

Success Looks Like: - ✅ Errors **serialize cleanly to JSON** with all context preserved - ✅ Integration with monitoring tools is **documented and works out-of-the-box** - ✅ Error metadata (severity, category) **maps to monitoring concepts** - ✅ They think "This makes our error tracking so much better"

Failure Looks Like: - ❌ Errors lose context when serialized - ❌ No guidance on monitoring integration - ❌ Custom metadata doesn't fit monitoring tools - ❌ They think "I'll write my own error handling"

1.4.3 @djed/http-client Success Criteria

1.4.3.1 The "No More Retry Logic" Test

Scenario: Developer calls unreliable external API

Success Looks Like: - ✅ Retry logic **works automatically** for transient failures (500, timeout) - ✅ Exponential backoff is **sensible and configurable** - ✅ Logs show **each retry attempt** clearly (via @djed/logger integration) - ✅ They think "I don't have to think about retries anymore"

Failure Looks Like: - ❌ Retries trigger on non-retriable errors (401, 404) - ❌ Backoff is too aggressive or too slow - ❌ No visibility into retry behavior - ❌ They think "I still need to write custom retry logic"

1.4.3.2 The "Debugging Bliss" Test

Scenario: Developer debugs failed API call

Success Looks Like: - ✅ Logs include **full request details** (URL, headers, body) via @djed/logger - ✅ Errors include **response details** (status, headers, body) via @djed/errors - ✅ Request IDs **flow through** for distributed tracing - ✅ They think "I can see exactly what was sent and received"

Failure Looks Like: - ❌ Logs are missing request/response details - ❌ Errors are generic ("Request failed") - ❌ No request correlation across services - ❌ They think "I have to add debug logging everywhere"

1.4.3.3 The "Ecosystem Harmony" Test

Scenario: Developer uses @djed/http-client with @djed/logger and @djed/errors

Success Looks Like: - ✅ Integration is **automatic** (just pass logger instance, errors work out-of-box) - ✅ Logs include **structured metadata** (duration, status, retry count) - ✅ Errors are **typed and actionable** (NetworkError, TimeoutError, etc.) - ✅ They think "All these packages were designed to work together"

Failure Looks Like: - ❌ Integration requires custom glue code - ❌ Logs are unstructured or missing data - ❌ Errors are generic JavaScript Error objects - ❌ They think "These packages don't know about each other"

1.5 🏗️ Phase 2B: Templates Success Criteria

1.5.1 mcp-server-minimal Template

1.5.1.1 The "15-Minute MCP Server" Test

Scenario: Developer needs to create a new MCP server

Success Looks Like: - ✅ They clone the template and have a **working MCP server in < 15 minutes** - ✅ Example tools are **realistic and instructive**, not just "hello world" - ✅ Documentation explains **how to add custom tools** step-by-step - ✅ They think "This template saved me hours of boilerplate"

Failure Looks Like: - ❌ Template doesn't run out-of-the-box - ❌ Examples are too trivial to learn from - ❌ No guidance on customization - ❌ They think "I should have started from scratch"

1.5.1.2 The "Best Practices Built-In" Test

Scenario: Junior developer builds their first MCP server from template

Success Looks Like: - ✅ Code structure **guides them toward good patterns** (separation of concerns, error handling) - ✅ Comments and docs **explain the "why"**, not just the "what" - ✅ Tests are **included and demonstrate testing patterns** - ✅ They think "I'm learning best practices just by using this template"

Failure Looks Like: - ❌ Code is poorly organized or uncommented - ❌ No explanation of design decisions - ❌ Tests are missing or not helpful - ❌ They think "I don't understand why it's structured this way"

1.5.1.3 The "Djed Integration Showcase" Test

Scenario: Developer sees how all Djed packages work together

Success Looks Like: - ✅ Template uses **@djed/logger, @djed/config, @djed/errors** seamlessly - ✅ Integration patterns are **obvious and well-commented** - ✅ They can **copy patterns to their own projects** confidently - ✅ They think "This is the reference implementation for Djed"

Failure Looks Like: - ❌ Template doesn't use Djed packages, or uses them poorly - ❌ Integration is hidden or unclear - ❌ Patterns don't generalize to other projects - ❌ They think "Why doesn't this use the Djed packages?"

1.5.2 express-api-starter Template

1.5.2.1 The "Production-Ready in 30 Minutes" Test

Scenario: Developer needs to start a new API project

Success Looks Like: - ✅ They run the template and have a **working API with auth, logging, error handling** in < 30 minutes - ✅ Examples include **realistic patterns** (pagination, validation, auth middleware) - ✅ Configuration is **environment-aware** (dev/staging/prod) - ✅ They think "I can deploy this to production after adding my business logic"

Failure Looks Like: - ❌ Template is missing critical features (auth, validation) - ❌ Examples are too simple (single GET endpoint) - ❌ No production considerations (security, monitoring) - ❌ They think "This is just a toy example"

1.5.2.2 The "Security by Default" Test

Scenario: Security engineer reviews template

Success Looks Like: - ✅ Security headers are **enabled by default** (helmet, CORS) - ✅ Input validation is **demonstrated** (request validation middleware) - ✅ Secrets are **managed properly** (via @djed/config, not hardcoded) - ✅ They think "Security is a first-class concern here"

Failure Looks Like: - ❌ Security is an afterthought - ❌ No input validation examples - ❌ Secrets are hardcoded or poorly managed - ❌ They think "This will get hacked in production"

1.6 🎓 Phase 2 Completion Success Criteria

1.6.1 The "Unified Ecosystem" Test

Scenario: Developer evaluates Djed as complete infrastructure solution

Success Looks Like: - ✅ They see **clear progression** from packages (building blocks) to templates (complete apps) - ✅ Documentation **links between packages** and shows integration patterns - ✅ All packages **share design philosophy** (progressive API, quality-first) - ✅ They think "This is a complete, well-designed ecosystem"

Failure Looks Like: - ❌ Packages feel disconnected - ❌ No guidance on how to use them together - ❌ Inconsistent quality or design - ❌ They think "This is just a random collection of packages"

1.6.2 The “LUXOR Standard” Test

Scenario: LUXOR team discusses infrastructure for new project

Success Looks Like: - ✅ Djed is the **default choice** (“Let’s use Djed for this”) - ✅ New team members are **pointed to Djed first** when starting projects - ✅ Internal projects **actively migrate to Djed** from ad-hoc solutions - ✅ They think “Djed is our infrastructure standard”

Failure Looks Like: - ❌ Djed is optional or unknown - ❌ Teams build custom solutions instead - ❌ No migration from existing projects - ❌ They think “Djed? What’s that?”

1.6.3 The “External Validation” Test

Scenario: External developer (outside LUXOR) discovers Djed

Success Looks Like: - ✅ They **understand Djed’s value** immediately (clear positioning, docs) - ✅ They **try it in a real project** (not just play around) - ✅ They **contribute back** (issues, PRs, suggestions) - ✅ They think “This is high-quality infrastructure worth using”

Failure Looks Like: - ❌ They’re confused about what Djed offers - ❌ They abandon after trying one package - ❌ No engagement or feedback - ❌ They think “This is just for LUXOR, not me”

1.7 📊 How to Measure Qualitative Criteria

1.7.1 User Interviews

Monthly: Talk to 3-5 developers using Djed - What do they love? - What frustrates them? - What’s missing?

1.7.2 Feedback Channels

- GitHub issues (feature requests, confusion, bugs)
- Internal Slack (questions, complaints, praise)
- npm reviews (if applicable)

1.7.3 Observation

- Watch new developers use Djed (pair programming, onboarding sessions)
- Note where they struggle, what they skip, what delights them

1.7.4 Self-Review

- **Monthly**: Re-read all documentation as if seeing it for the first time
- **Quarterly**: Build a sample project using only public docs (no insider knowledge)

1.8 🎯 Success Criteria for This Document

This document itself succeeds if: - ✅ Team references it when making design decisions - ✅ Code reviews cite criteria ("Does this pass the 5-minute test?") - ✅ Retrospectives use it to evaluate what worked/didn't - ✅ New contributors understand the quality bar

This document fails if: - ❌ It's written once and never referenced - ❌ Team doesn't agree with the criteria - ❌ Criteria are too vague to be actionable - ❌ It becomes a checklist without understanding the "why"

1.9 🌟 The North Star

Every Djed package and template should make developers think:

> *"Whoever built this really cares about my experience. This is infrastructure I can trust."*

If we achieve that feeling consistently, we've succeeded—regardless of download numbers.

Created: 2025-11-03 **Status**: Living document (update based on real-world feedback) **Owner**: Djed Core Team