NexusLink Codex Agent Waterfall QA Prompt File

Overview

This agent prompt document instructs Codex on building `nlink.exe` from modular `.c/.h` source trees using semantic intent recognition, AST validation, waterfall QA methodology, and state machine minimization. Each section represents a task in a waterfall system with formal soundness proofs.

Critical Context: You are working on the OBINexus NexusLink project—a build orchestration system that enforces zero false-positive eliminations through formal QA quadrant validation.

🚀 Complete Build Flow Example

Full Command Line Invocation:

```
# Standard build (beta channel for development)
nlink.exe -S . -B build/ --channel beta

# Alpha channel for production
nlink.exe -S . -B build/ --channel alpha --production

# With custom manifest and validation
nlink.exe -S src/ -B out/ --manifest custom.nlink.in --qa-validate

# Minimal production build
nlink.exe -S . -B release/ --channel alpha -DMINIMAL_BUILD=ON
```

Build Flow Sequence:

```
1. Read pkg.nlink.in (or --manifest override)
   ↓
2. Apply whitelist/blacklist filters:
   - Include: **/*.c, **/*.h, **/nlink.txt
   - Exclude: **/tmp/*, **/*.log, patterns from blacklist
   ↓
3. Evaluate scripting conditionals based on channel:
   - Beta: Enable all features
   - Alpha: Production features only
   ↓
4. Generate pkg.nlink.in.xml (validated)
   ↓
5. Process filtered module nlink.txt files
   ↓
6. Build expression transformation graph:
   - tokenizer/lexer/statement.c → expression.c
   - Apply statement filtering (remove debug in alpha)
   ↓
```

```
7. Apply state machine minimization
   ↓
8. Run waterfall QA validation:
   - Ensure Beta APIs not exposed in Alpha
   ↓
9. Generate channel-specific outputs:
   - build/bin/nlink.exe (tagged with channel)
   - build/lib/libnlink-{channel}.so
   - build/cache/ (whitelisted artifacts only)
```

Channel Isolation Example:

```
// In tokenizer/advanced.c
#ifdef BETA_BUILD
void experimental_tokenizer_api() {
    // Only available in beta channel
}
#endif

// In parser/core.c
void stable_parser_api() {
    // Available in both beta and alpha
#ifdef ALPHA_BUILD
    // Strict validation in production
    validate_input_strictly(input);
#endif
}
```

Module Resolution with Filtering:

```
Given source tree:
src/
  tokenizer/
    lexer/
      statement.c     ✓ (whitelisted: **/*.c)
      test_lexer.c    ✗ (blacklisted: **/test_*)
      debug.log       ✗ (blacklisted: **/*.log)
    nlink.txt         ✓ (whitelisted: **/nlink.txt)
  tmp/
    work.c            ✗ (blacklisted: **/tmp/*)

After filtering:
- Process: tokenizer/lexer/statement.c
- Process: tokenizer/nlink.txt
- Skip: Everything else
```

Statement → Expression Problem Analysis:

```
// Example: Detecting feature duplication in statement handling
// Check nlink_qa_poc/src/ for existing patterns:

// BAD: Reimplementing existing logic
if (token_type == TOKEN_IF) {
    // This might already exist in nlink_cli/lexer/
    handle_if_statement();  // ❌ Check for duplication first!
}

// GOOD: Reuse existing implementation
#include "nlink_cli/lexer/statement_handler.h"
if (token_type == TOKEN_IF) {
    // Reuse from existing POC
    nlink_handle_statement(token_type);  // ✓ No duplication
}
```

Quality Assurance Code Inspection:

```
For each .c/.h file pair:
1. Identify decision points (if/else/switch)
2. Map to QA quadrants:

   // In parser.c
   if (validate_syntax(input)) {      // Decision point
       process_ast(input);            // TP: Valid input processed
   } else {
       report_error(input);           // TN: Invalid input rejected
   }

3. Ask critical questions:
   - Can valid input be rejected? (FN risk)
   - Can invalid input be accepted? (FP risk)
   - Are all edge cases covered?

4. Check POC implementations:
   - nlink_qa_poc/test/unit/
   - nlink_qa_poc/test/integration/
   - Look for similar test patterns
```

Feature/Expression Mapping Verification:

```
Before implementing any expression transformation:

1. Check existing mappings in POC folders:
   find nlink*/ -name "*.c" -exec grep -l "expression" {} \;

2. Verify transformation patterns:
   - statement.c → expression.c already exists?
```

```
   - Transformation rules already defined?
   - QA validation already implemented?

3. Document transformation intent:
   // INTENT: Transform imperative to functional
   // EXISTING: Check nlink_enhanced/transforms/
   // REUSE: Expression evaluator from nlink_lazy/

4. Apply QA heuristics to transformations:
   - Input statement correctly parsed? (TP)
   - Invalid statement rejected? (TN)
   - Valid statement missed? (FN)
   - Invalid statement accepted? (FP)
```

Codex Duplication Prevention Workflow:

```
on_feature_request:
  - name: "Check POC implementations"
    steps:
      - search: "grep -r 'feature_keyword' nlink*/"
      - analyze: "Read matching files for implementation"
      - compare: "Check if functionality exists"

  - name: "Analyze existing QA patterns"
    steps:
      - locate: "Find test files in nlink_qa_poc/"
      - extract: "Identify TP/TN/FP/FN patterns"
      - reuse: "Apply existing QA heuristics"

  - name: "Decision matrix"
    conditions:
      - if: "Feature exists in POC"
        then: "Extend/refactor existing code"
      - elif: "Similar pattern exists"
        then: "Adapt pattern to new use case"
      - else: "Implement with QA validation"
```

Remember: Every line of new code should be checked against existing POC implementations. The poc/ folder contains battle-tested patterns that should be reused, not reimplemented.

🗀 Existing POC Inventory

Critical: Check These Before Any Implementation

| Project | Purpose | Check For | |### 🔬 Real Example: Feature Implementation with POC Check

Scenario: Implementing a new parser feature

```
# Step 1: Check if parser logic exists
$ grep -r "parse_expression" nlink*/
nlink_cli/src/parser.c:142: int parse_expression(token_t* token)
nlink_enhanced/parser/expr.c:89: result = parse_expression_tree(ast);
nlink_qa_poc/test/unit/test_parser.c:23: test_parse_expression();

# Step 2: Examine existing implementation
$ cat nlink_cli/src/parser.c | grep -A 20 "parse_expression"
```

```
// Found in nlink_cli/src/parser.c:
int parse_expression(token_t* token) {
    if (token->type == TOKEN_NUMBER) {      // TP: Valid number parsed
        return parse_number(token);
    } else if (token->type == TOKEN_ID) {   // TP: Valid identifier parsed
        return parse_identifier(token);
    } else {                                // TN: Invalid token rejected
        return PARSE_ERROR;
    }
}

// Step 3: Instead of reimplementing, REUSE:
#include "nlink_cli/src/parser.h"

// Step 4: Extend with QA validation:
int enhanced_parse_expression(token_t* token) {
    // QA: Check preconditions (prevent FP)
    if (!token || !token->data) {
        log_qa_event("NULL token", QA_FALSE_POSITIVE_PREVENTED);
        return PARSE_ERROR;
    }

    // Reuse existing, tested implementation
    int result = parse_expression(token);

    // QA: Validate postconditions
    if (result == PARSE_SUCCESS) {
        validate_expression_tree();  // Prevent FN
    }

    return result;
}
```

QA Decision Tree for Code Reuse:

```
Feature_Request: "Parse expression with validation"
│
├─ Check_POC_Folders:
│  ├─ Found in nlink_cli/? → YES
```

```
│   │   └─ Action: REUSE and EXTEND
│   │
│   └─ Not found? → NO
│      └─ Check_Similar_Patterns:
│         ├─ Found similar? → YES
│         │  └─ Action: ADAPT pattern
│         └─ Nothing similar? → NO
│            └─ Action: NEW implementation with QA

QA_Validation_Required:
  - Every if: "What if condition is true?" → TP or FN?
  - Every else: "What if condition is false?" → TN or FP?
  - Every function: "Check poc/ for existing implementation"
```

Statement → Expression Transformation Verification

Before Implementing ANY Transformation:

```
# 1. Check for existing statement handlers
find nlink*/ -path "*/lexer/*" -name "statement.c" -o -name "*statement*"

# 2. Check for existing expression evaluators
find nlink*/ -name "*expression*" -o -name "*expr*"

# 3. Look for transformation patterns
grep -r "statement.*expression" nlink*/
grep -r "transform" nlink_enhanced/
```

Example: If-Statement to Conditional Expression

```
// CHECK FIRST: Does this already exist?
// nlink_enhanced/transforms/stmt_to_expr.c might have:

// Statement form (imperative):
if (x > 0) {
    result = positive_value;
} else {
    result = negative_value;
}

// Expression form (functional):
result = (x > 0) ? positive_value : negative_value;

// QA Analysis Required:
// - TP: Correct transformation preserves semantics
// - TN: Invalid statement correctly rejected
// - FP: Wrong transformation changes behavior [CRITICAL]
// - FN: Valid transformation opportunity missed
```

Transformation QA Checklist:

```
□ Check nlink_lazy/ for deferred evaluation patterns
□ Check nlink_enhanced/ for optimization transforms
□ Check nlink_simplified/ for basic implementations
□ Verify semantic equivalence (no FP transformations)
□ Test edge cases from nlink_qa_poc/test/
□ Document why transformation is safe
```

Code Quality Enforcement:

```c
// For EVERY new function, add QA markers:
int transform_statement_to_expression(ast_node_t* stmt) {
    // QA_CHECK: Does nlink_enhanced already have this?
    // REUSE_FROM: nlink_enhanced/transforms/ast_transform.c

    if (!stmt) {
        // QA_QUADRANT: TN - Correctly reject null input
        return TRANSFORM_ERROR;
    }

    if (stmt->type == AST_IF_STATEMENT) {
        // QA_QUADRANT: TP - Valid transformation case
        // CHECK_POC: nlink_lazy/expr_eval.c line 234
        return transform_if_to_conditional(stmt);
    }

    // QA_QUADRANT: TN - Correctly reject non-transformable
    return TRANSFORM_NOT_APPLICABLE;
}
```

Final Rule: No code without POC check. No branch without QA analysis. No feature without duplication verification.

🌐 Task 8: Glob Pattern Support & Path Resolution

CLI Path Pattern Handling (Like CMake)

The `nlink.exe` CLI must support flexible glob patterns for source discovery and build orchestration:

Supported Glob Patterns:

```
# Basic patterns
nlink -S .                    # Current directory
nlink -S src/                 # Specific directory
nlink -S **/*.c              # All C files recursively
```

```
nlink -S **/tokenizer/*.c    # All C files in any tokenizer directory
nlink -S src/**/main.c       # Find all main.c files under src/

# Complex patterns (enhanced build processing)
nlink -S "src/**/*.{c,h}"    # All C and H files
nlink -S "**/cli/main.c"     # Entry point discovery
nlink -S "**/{tokenizer,parser}/*.c"  # Multiple module patterns

# Feature-based patterns
nlink -S . --feature-glob "feature_*/**/*.c"
nlink -S . --exclude-glob "**/test_*.c"
```

Path Resolution in pkg.nlink.in:

```
# Glob-based source discovery
glob_sources(TOKENIZER_SOURCES
    src/tokenizer/**/*.c
    src/tokenizer/lexer/*.c
    !src/tokenizer/test_*.c    # Exclude test files
)

glob_headers(TOKENIZER_HEADERS
    include/tokenizer/**/*.h
    !include/tokenizer/internal/*.h  # Exclude internal headers
)

# Feature intention with base cli/main.c
declare_entry_points()
    primary(cli/main.c)              # Base entry point
    feature(tools/nlink_tool.c)      # Additional tools
    glob(cli/*_main.c)               # Discover all CLI variants
enddeclare()

# Complex build pattern matching
if(ENABLE_ALL_MODULES)
    glob_sources(ALL_SOURCES
        src/**/*.c                   # All source files
        !src/**/deprecated/*.c       # Except deprecated
        !src/**/experimental/*.c     # Except experimental (unless beta)
    )
endif()
```

Agent Instructions for Glob Processing:

```
1. Parse glob patterns using POSIX glob() or custom matcher:
   - Support ** for recursive directory matching
   - Support * for single-level wildcards
   - Support ? for single character matching
   - Support [...] for character classes
```

```
   - Support {...} for alternation

2. Implement exclusion patterns (! prefix):
   - Process includes first
   - Apply exclusions to filter results
   - Maintain order of operations

3. Path normalization:
   - Convert all paths to canonical form
   - Resolve .. and . components
   - Handle both / and \ separators

4. Feature-aware globbing:
   - Tag files by feature based on path
   - Apply channel restrictions to glob results
   - Respect module boundaries
```

CLI Command Structure with Globs:

```
# Source specification with enhanced patterns
nlink -S <glob-pattern> [options]

# Examples:
nlink -S "**/*.c" -B build/                    # All C files
nlink -S "src/**/*.c" --exclude "**/test_*"    # Exclude tests
nlink -S . --source-glob "**/*.c" --header-glob "**/*.h"

# Multiple source patterns
nlink -S src/ -S lib/ -S tools/               # Multiple roots
nlink -S "src/**/*.c;lib/**/*.c"              # Semicolon-separated

# Feature-based selection
nlink -S . --feature tokenizer --feature parser
# Equivalent to: -S "**/tokenizer/**/*.[ch]" -S "**/parser/**/*.[ch]"
```

Glob Resolution Algorithm:

```
// Pseudo-code for glob resolution in nlink
typedef struct {
    char** patterns;          // Array of glob patterns
    char** exclusions;        // Array of exclusion patterns
    bool recursive;           // Support ** patterns
    bool case_sensitive;      // Platform-specific
} glob_config_t;

// Resolution process
list_t* resolve_glob_patterns(glob_config_t* config) {
    list_t* files = list_create();
```

```
    // Phase 1: Expand inclusion patterns
    for (pattern in config->patterns) {
        if (has_double_star(pattern)) {
            expand_recursive(pattern, files);
        } else {
            expand_simple_glob(pattern, files);
        }
    }

    // Phase 2: Apply exclusions
    for (exclusion in config->exclusions) {
        filter_matching(files, exclusion);
    }

    // Phase 3: Normalize and deduplicate
    normalize_paths(files);
    remove_duplicates(files);

    return files;
}
```

Integration with Build System:

```
# In pkg.nlink.in - using glob patterns
configure_sources()
    # Base patterns
    base_glob("src/**/*.c")

    # Module-specific patterns
    module_glob(tokenizer "src/tokenizer/**/*.[ch]")
    module_glob(parser "src/parser/**/*.[ch]")
    module_glob(cli "src/cli/**/*.[ch]")

    # Conditional patterns
    if(INCLUDE_EXAMPLES)
        add_glob("examples/**/*.c")
    endif()

    # Always exclude
    exclude_glob("**/*.tmp")
    exclude_glob("**/build/**")
    exclude_glob("**/.git/**")
endconfigure()
```

Path Pattern Best Practices:

```
1. Always use forward slashes in patterns (cross-platform)
2. Quote patterns containing spaces or special chars
3. Use ** sparingly (performance impact)
```

```
4. Prefer specific patterns over broad ones
5. Document exclusion reasons in comments
6. Test patterns with --dry-run flag

# Debugging glob patterns
nlink -S "**/*.c" --dry-run --verbose
# Shows: Files that would be included without building
```

This glob support ensures that `nlink.exe` can handle complex build scenarios with flexible source discovery, similar to CMake but with enhanced pattern matching for the statement→expression transformation pipeline.

Unlocked Command Enhancement:

Unlike rigid build systems, `nlink.exe` commands are **not locked** but can be enhanced through:

```
# Base command structure (unlocked)
nlink -S <source> -B <build> [enhancements...]

# Enhancement examples:
nlink -S **/*.c \
    --enhance-with "custom_transform.nlink" \
    --plugin "optimization_pass.so" \
    --override-pattern "src/special/**/*.c" \
    --custom-glob-engine "pcre2"

# Feature intention with cli/main.c as base
nlink -S . \
    --entry cli/main.c \
    --feature-origin "src/features/" \
    --feature-intent "experimental:tokenizer,stable:parser"

# Complex build with multiple intentions
nlink -S src/**/*.c \
    --base-entry cli/main.c \
    --alt-entry "tools/tool_main.c" \
    --feature-glob "feature_*/**/*.c" \
    --intent-map "intent.yaml"
```

Feature Intention & Origin Mapping:

```
# intent.yaml - Feature intention configuration
base:
    entry: cli/main.c
    stability: stable

features:
    tokenizer:
        origin: src/tokenizer/
        intent: transform_statements
```

```
        glob: "tokenizer/**/*.[ch]"
        channel: beta

    parser:
        origin: src/parser/
        intent: build_ast
        glob: "parser/**/*.[ch]"
        channel: alpha

    experimental:
        origin: src/experimental/
        intent: research_features
        glob: "experimental/**/*.[ch]"
        channel: beta
        exclude: true  # Not in alpha builds
```

Glob CLI Usage in Build Process:

```
# Check what files would be included (POC verification)
nlink -S "**/*.c" --list-files --check-poc

# Output shows POC matches:
# src/tokenizer/lexer.c [EXISTS in nlink_cli/lexer.c]
# src/parser/ast.c [NEW]
# src/cli/main.c [BASE ENTRY]

# Enhanced glob with feature detection
nlink -S . --detect-features --glob-enhance

# Automatically detects:
# - Feature modules by directory structure
# - Entry points by main() detection
# - Test files by naming patterns
# - POC duplicates by similarity analysis
```

Command Enhancement API:

```
// Commands can be enhanced at runtime
typedef struct {
    glob_handler_t custom_glob;      // Custom glob engine
    transform_fn_t pre_transform;    // Pre-processing
    validate_fn_t qa_validator;      // QA enforcement
    pattern_fn_t pattern_enhancer;   // Pattern expansion
} command_enhancer_t;

// Register enhancement
nlink_register_enhancement("my_enhance", enhancer);
```

```
// Use in CLI
// nlink -S . --enhance my_enhance
```

This ensures the CLI remains flexible and can be enhanced for complex build scenarios without being locked into rigid patterns.

---------|---------|----------| | `nlink_cli/` | CLI interface & configuration parser | Command parsing, config handling | | `nlink_cli_semverx/` | Semantic versioning implementation | Version parsing, compatibility checks | | `nlink_enhanced/` | Enhanced features & optimizations | Advanced algorithms, performance code | | `nlink_lazy/` | Lazy evaluation patterns | Deferred computation, memoization | | `nlink_qa_poc/` | Quality assurance framework | Test patterns, validation logic | | `nlink_symbols/` | Symbol table management | Symbol resolution, versioning | | `nlink_simplified/` | Simplified/minimal implementation | Core algorithms, basic patterns |

POC Code Reuse Matrix:

```
Feature Request → POC to Check → Reusable Components

Configuration parsing → nlink_cli/ → config.c, parser_interface.c
Version handling → nlink_cli_semverx/ → semver.c, version_compare.c
Symbol management → nlink_symbols/ → symbol_table.c, resolution.c
QA validation → nlink_qa_poc/ → test frameworks, quadrant analysis
Performance opt → nlink_enhanced/ → optimization patterns
Lazy loading → nlink_lazy/ → deferred evaluation logic
```

Pre-Implementation Checklist:

```
# Before writing ANY new code, run these checks:

# 1. Search for existing implementations
find . -name "*.c" -o -name "*.h" | xargs grep -l "your_feature"

# 2. Check POC test coverage
find nlink_qa_poc/test -name "*.c" | xargs grep "test_pattern"

# 3. Look for similar algorithms
grep -r "algorithm_name" nlink_enhanced/ nlink_simplified/

# 4. Verify no duplicate symbols
find . -name "*.c" | xargs grep "function_name"
```

Task 7: Whitelist/Blacklist Manifest Layer

Build Artifact Control (Similar to .gitignore):

The manifest layer supports whitelist/blacklist patterns for controlling what gets built, cached, and included in final outputs.

In pkg.nlink.in:

```
# Whitelist: Explicitly include for caching
configure_whitelist()
    # Source patterns to always process
    pattern("**/*.c")           # All C source files
    pattern("**/*.h")           # All headers
    pattern("**/nlink.txt")     # All module configs
    pattern("**/*.nlink")       # All sub-configs

    # Cache generation rules
    cache_artifacts("**/*.o")    # Object files
    cache_artifacts("**/*.a")    # Static libraries
endconfigure()

# Blacklist: Exclude from build/output
configure_blacklist()
    # Temporary/generated files (like .gitignore)
    pattern("**/tmp/*")
    pattern("**/*.tmp")
    pattern("**/*~")            # Editor backups
    pattern("**/*.swp")         # Vim swap files

    # Build artifacts to exclude
    pattern("**/test_*")        # Test executables
    pattern("**/debug_*")       # Debug builds
    pattern("**/*.log")         # Log files

    # Conditional exclusions
    if(MINIMAL_BUILD)
        pattern("**/examples/*")
        pattern("**/benchmarks/*")
    endif()

    # Macro-based exclusions
    if(NOT ENABLE_DEPRECATED)
        pattern("**/deprecated/*")
        macro_exclude(DEPRECATED_API)
    endif()
endconfigure()

# Expression-based intent
declare_build_intents()
    # Specify where outputs go
    intent(EXECUTABLES → "${BUILD_DIR}/bin/*.exe")
    intent(LIBRARIES → "${BUILD_DIR}/lib/*.{so,a}")
    intent(OBJECTS → "${BUILD_DIR}/obj/**/*.o")

    # Source organization intent
```

```
    intent(CORE_MODULES → "src/core/**/")
    intent(FEATURE_MODULES → "src/features/**/")
    intent(THIRD_PARTY → "external/**/")
endeclare()
```

Path Resolution for pkg.nlink.in:

```
# Specify manifest search paths
set(NLINK_SEARCH_PATHS
    "${CMAKE_CURRENT_SOURCE_DIR}/pkg.nlink.in"
    "${CMAKE_CURRENT_SOURCE_DIR}/config/pkg.nlink.in"
    "${PROJECT_ROOT}/pkg.nlink.in"
)

# Build output structure
configure_build_layout()
    # Final structure:
    # build/
    #   ├── bin/         # *.exe executables
    #   ├── lib/         # *.so, *.a libraries
    #   ├── obj/         # *.o object files
    #   ├── include/     # Exported headers
    #   └── cache/       # Build cache

    set(BUILD_ROOT "${PROJECT_BINARY_DIR}")
    set(INSTALL_PREFIX "${BUILD_ROOT}/install")
endconfigure()
```

Statement/Macro Exclusion:

```
# Exclude code blocks via macros
macro_blacklist()
    # Exclude experimental features in alpha
    if(CHANNEL STREQUAL "alpha")
        define(EXCLUDE_EXPERIMENTAL)
    endif()

    # In source code:
    # #ifndef EXCLUDE_EXPERIMENTAL
    #     void experimental_feature() { ... }
    # #endif
endmacro()

# Statement-level exclusion
statement_filter()
    # Remove debug statements in production
    if(PRODUCTION_BUILD)
        remove_statements("assert(*)")
        remove_statements("debug_print(*)")
```

```
        remove_statements("TRACE_*")
    endif()
endstatement()
```

Agent Instructions:

```
1. Parse whitelist/blacklist from pkg.nlink.in:
   - Apply patterns in order (blacklist overrides whitelist)
   - Respect conditional blocks based on build config

2. Filter source tree:
   - Include only whitelisted patterns
   - Exclude all blacklisted patterns
   - Apply macro-based filtering

3. Process build intents:
   - Map source patterns to output destinations
   - Organize by intent (executables, libraries, etc.)

4. Generate filtered build graph:
   - Only include non-blacklisted sources
   - Apply statement-level filtering
   - Respect macro exclusions

5. Cache management:
   - Cache whitelisted artifacts
   - Never cache blacklisted patterns
   - Implement cache invalidation rules
```

🔧 Task 1: Manifest Extraction & Scripting Layer

Input: `pkg.nlink.in` (scripting manifest)
Schema: `nlink.schema.xml`
Output: `pkg.nlink.in.xml` → `pkg.nlink`

The `pkg.nlink.in` is a CMake-like scripting manifest that controls all inputs across the `nlink/` root domain. It uses an extended scripting syntax for semantic configuration:

pkg.nlink.in Scripting Syntax:

```
# Project declaration
project(NexusLink VERSION 1.0.0)

# Global configuration
set(BUILD_CHANNELS "experimental;stable")
set(DEFAULT_SCOPE "protected")
```

```
# Conditional module inclusion
if(ENABLE_CRYPTO)
    add_module(crypto SCOPE protected)
    set(SHANNON_ENTROPY_MIN 7.0)
endif()

# Expression mappings
declare_expressions()
    map(statement → expression)
    map(imperative → declarative)
    map(procedure → function)
enddeclare()

# semverx configuration
configure_semverx()
    channel(experimental VERSION_SUFFIX "-alpha")
    channel(stable LOCK_COMPATIBILITY true)
    if(PRODUCTION_BUILD)
        enforce_compatibility(STRICT)
    endif()
endconfigure()

# Build output configuration
configure_output()
    binary_dir(${BUILD_DIR}/bin)
    library_dir(${BUILD_DIR}/lib)
    object_dir(${BUILD_DIR}/obj)

    if(MINIMAL_BUILD)
        strip_symbols(AGGRESSIVE)
        optimize_size(MAXIMUM)
    endif()
endconfigure()
```

Processing Pipeline:

1. **pkg.nlink.in** → Scripting source with conditionals
2. **pkg.nlink.in.xml** → Resolved XML manifest after script evaluation
3. **pkg.nlink** → Final flattened build configuration

Agent Instructions:

```
1. Parse pkg.nlink.in using extended CMake-like parser:
   - Evaluate all conditionals based on environment
   - Resolve variable substitutions
   - Process nested function calls

2. Transform to XML intermediate format:
   - Convert script directives to XML elements
   - Preserve evaluation context as attributes
   - Validate against nlink.schema.xml
```

```
3. Generate final pkg.nlink:
   - Flatten all resolved configurations
   - Compute module dependency graph
   - Apply semverx version locking

4. Process all *.nlink files in source tree:
   - Each follows same scripting syntax
   - Inherit from parent configurations
   - Override with local settings

5. Validate semantic consistency:
   - No channel cross-contamination
   - All exports properly scoped
   - Version compatibility enforced
```

Validation Checkpoint: Ensure zero schema violations and complete channel separation.

Task 2: Module Parsing with Expression Intent

Detailed Module Structure:

```
src/
├── tokenizer/
│   ├── lexer/
│   │   ├── statement.c    → expression.c  [Expression mapping]
│   │   ├── scanner.c      → tokens.c
│   │   └── buffer.c       → stream.c
│   ├── include/
│   │   └── tokenizer.h
│   └── nlink.txt          → Module configuration
├── parser/
│   ├── ast.c
│   ├── include/parser.h
│   └── nlink.txt
├── cli/
│   ├── main.c             → Entry point for nlink.exe
│   ├── commands/*.c
│   └── nlink.txt
└── common/
    ├── utils.c
    └── nlink.txt
```

Expression Layer Mapping:

- **Statement** → **Expression**: Transform imperative to declarative
- **Example**: `statement.c` (if/while/for) → `expression.c` (true|false evaluation)
- **Module Intent**: Derived from nlink.txt declarations
- **Build Graph**: Source → Expression → Compilation Unit

nlink.txt Extended Syntax (CMake-like):

```
# Module configuration with scripting
if(FEATURE_ADVANCED_LEXER)
    add_source(lexer/advanced.c)
    add_expression(lexer/statement.c → expression.c)
endif()

declare_module(tokenizer)
    scope(public)
    version(1.0.0)

    if(BUILD_TYPE STREQUAL "Debug")
        add_definitions(-DDEBUG_LEXER)
    endif()

    exports(
        tokenize → public
        lex_init → protected
        internal_state → private
    )
enddeclare()
```

CLI Command Structure:

```
# Basic invocation
nlink.exe -S . -B build/

# With manifest override
nlink.exe -S . -B build/ --manifest pkg.nlink.in

# Options:
# -S <path>    Source directory (default: .)
# -B <path>    Build directory (default: build/)
# --manifest   Override manifest file
# --validate   Run QA validation only
```

Agent Instructions:

```
For each module directory:
1. Parse nlink.txt using CMake-like syntax parser:
   - Handle if()/endif() conditionals
   - Process declare_module() blocks
   - Evaluate expressions based on build config

2. Map expression transformations:
   - Identify statement.c → expression.c patterns
   - Build transformation graph
```

```
   - Validate expression completeness

3. Process nested structures (e.g., tokenizer/lexer/):
   - Recursively parse subdirectories
   - Maintain hierarchy in symbol table
   - Resolve relative includes

4. Link implementations with declarations:
   - Match *.c files with corresponding *.h
   - Verify expression mappings exist
   - Tag symbols with proper scope

5. Build consolidated module graph:
   - cli/main.c as primary entry point
   - Resolve inter-module dependencies
   - Generate build order based on deps
```

Critical: The expression layer must maintain semantic equivalence during transformation.

🧠 Task 3: AST Optimization & State Machine Minimization

Based on state machine minimization principles from the OBINexus methodology:

Minimization Algorithm:

```
1. Build complete AST from parsed modules
2. Apply state machine minimization:
   - Identify all reachable states from entry points
   - Mark unreachable states for elimination
   - Validate transition integrity
3. Compute minimal symbol set preserving functionality
4. Generate optimized AST with dead code eliminated
```

Agent Instructions:

```
1. Construct compiler graph from all modules
2. Perform reachability analysis from main() and exported symbols
3. Apply AST transformations:
   - Node reduction (eliminate unreachable)
   - Path optimization (minimize state checks)
   - Memory efficiency (reduce allocations)
4. Preserve all symbols marked as "protected" regardless of reachability
```

⚙️ Task 4: Waterfall QA with Formal Soundness

Command: `nlink.exe --qa-validate`

QA Quadrant Validation (Zero FP Tolerance):

```
True Positive (TP):  Correctly eliminated dead code ✓
True Negative (TN):  Correctly retained live code ✓
False Positive (FP): Incorrectly eliminated live code ✗ [CRITICAL ERROR]
False Negative (FN): Incorrectly retained dead code [OPTIMIZATION MISS]
```

Agent Instructions:

```
1. Execute minimization on test corpus
2. For each elimination decision:
   - Classify into TP/TN/FP/FN quadrant
   - Log decision rationale with symbol graph proof
3. ABORT if any FP detected (zero tolerance policy)
4. Generate quadrant report with metrics:
   - FP rate: MUST be 0.0%
   - TP rate: Target ≥ 95.0%
   - FN rate: Maximum 5.0%
5. Run 3 verification iterations for soundness
```

Formal Proof Required: Each elimination must include reachability proof from entry points.

☑ Task 5: Build Output with Crypto Validation

Output Structure:

```
build/
  bin/        → Executables (crypto-signed)
  lib/        → Libraries (version-locked)
  obj/        → Object files (cached)
```

Agent Instructions:

```
For each output artifact:
1. Verify module passed QA (check quadrant report)
2. Validate against pkg.nlink manifest declarations
3. Apply .nlinkignore exclusions
4. Compute Shannon entropy for crypto validation:
   - Minimum entropy threshold: 7.0 bits
   - Chi-square test for distribution
5. Generate metadata:
   - Version info (semverx format)
```

```
    - Build reproducibility hash
    - Symbol export table
```

Task 6: Semantic Versioning & Dual-Channel Architecture

Primary Channel Model:

```
Beta Channel:  New features, experimental APIs, developer testing
Alpha Channel: Production-ready, stable APIs, customer deployments
```

semverx Extended Format:

```
major.minor.patch-channel.build+metadata
Examples:
  Beta:  1.0.0-beta.42+sha256.abc123    (feature development)
  Alpha: 1.0.0-alpha.1+sha256.def456    (production release)
```

Channel Usage Flow:

```
Developer Workflow:
  1. Develop in Beta → Test features
  2. Promote to Alpha → Production use
  3. Never expose Beta APIs in Alpha builds

Similar to Git:
  Beta  = feature branches
  Alpha = main/production branch
```

Channel Configuration in pkg.nlink.in:

```
# Channel definitions (extensible)
define_channels()
    channel(beta
        PURPOSE "New feature development"
        STABILITY "experimental"
        FLAGS "-DBETA_BUILD -DENABLE_EXPERIMENTAL"
    )

    channel(alpha
        PURPOSE "Production deployments"
        STABILITY "stable"
        FLAGS "-DALPHA_BUILD -DSTRICT_COMPATIBILITY"
    )
```

```
    # Schema allows adding custom channels
    channel(canary
        PURPOSE "Early access testing"
        STABILITY "unstable"
        FLAGS "-DCANARY_BUILD"
    )
enddefine()

# Channel-specific rules
if(CHANNEL STREQUAL "beta")
    enable_all_features()
    set(DEPRECATION_WARNINGS OFF)
elseif(CHANNEL STREQUAL "alpha")
    enable_stable_features_only()
    set(STRICT_API_CHECKING ON)
    set(DEPRECATION_WARNINGS ERROR)
endif()
```

Agent Instructions:

```
1. Parse version from pkg.nlink.in.xml:
   - Extract channel identifier
   - Determine feature set based on channel

2. Apply channel-specific compilation:
   - Beta: All features enabled, relaxed checking
   - Alpha: Stable features only, strict validation
   - Custom: As defined in schema

3. Enforce channel isolation:
   - Beta symbols NEVER exposed to Alpha
   - Use #ifdef guards for channel-specific code
   - Generate separate symbol tables per channel

4. Version metadata generation:
   - Embed channel in all outputs
   - Include build timestamp
   - Add reproducibility hash
```

☠️ Failsafe & Error Recovery

On QA Failure:

```
1. Immediately halt build process
2. Rollback to last known good state
3. Generate detailed failure report:
   - Offending symbol(s)
```

```
    - Elimination decision graph
    - Quadrant classification
4. Write to pkg.nlink.error.log with:
    - Timestamp
    - Build configuration
    - Full symbol dependency trace
5. Exit with non-zero status code
```

On Manifest Violation:

```
1. Log schema validation errors
2. Highlight conflicting directives
3. Suggest resolution based on intent patterns
4. Require manual intervention
```

Agent TODO Checklist

Phase 1: Preparation

- [ ] Validate `pkg.nlink.in` exists and is readable
- [ ] Load `nlink.schema.xml` for validation
- [ ] Check for `.nlinkignore` in project root
- [ ] Initialize ETPS telemetry for build tracking

Phase 2: Parsing

- [ ] Parse `pkg.nlink.in` → generate `pkg.nlink.in.xml`
- [ ] Validate XML against schema (zero violations)
- [ ] Transform XML → `pkg.nlink` manifest
- [ ] Parse all module `nlink.txt` files
- [ ] Build complete symbol inventory

Phase 3: Analysis

- [ ] Construct compiler graph from sources
- [ ] Link `.c/.h` pairs by expression intent
- [ ] Apply state machine minimization
- [ ] Compute reachability from entry points
- [ ] Generate elimination candidate set

Phase 4: Validation

- [ ] Run waterfall QA with quadrant classification
- [ ] Verify FP rate = 0.0% (abort if violated)
- [ ] Generate formal proofs for eliminations
- [ ] Compute Shannon entropy for crypto modules
- [ ] Validate chi-square distribution

Phase 5: Build

- [ ] Generate optimized binaries to `build/bin`
- [ ] Create version-locked libraries in `build/lib`
- [ ] Apply channel-based exposure rules
- [ ] Sign artifacts with build metadata
- [ ] Generate minimization report

Phase 6: Verification

- [ ] Re-run QA validation on outputs
- [ ] Verify symbol preservation
- [ ] Check binary size reduction (≥40% target)
- [ ] Validate metadata integrity
- [ ] Generate final build attestation

📋 Critical Reminders

1. **Zero False Positives**: Any FP immediately fails the build. No exceptions.
2. **Intent Over Inference**: Never guess module intent—only use explicit configuration.
3. **Channel Isolation**: Alpha features must NEVER leak to stable builds.
4. **Formal Proofs**: Every elimination requires a reachability proof.
5. **Crypto Validation**: All security modules require entropy validation.

Example Module Configuration Hierarchy

Root `pkg.nlink.in`:

```
# Global project configuration
project(NexusLink VERSION 1.0.0)
set(DEFAULT_SCOPE protected)
set(ENABLE_CRYPTO ON)

# Global expression mappings
declare_global_expressions()
    pattern(statement.c → expression.c)
    pattern(imperative.c → functional.c)
enddeclare()
```

`src/tokenizer/nlink.txt`:

```
inherit_from(../../pkg.nlink.in)

declare_module(tokenizer)
    scope(public)  # Override default
    version(1.0.0)
```

```
    # Tokenizer-specific expressions
    add_expression(lexer/statement.c → expression.c)
    add_expression(lexer/scanner.c → tokens.c)

    if(FEATURE_UNICODE)
        add_source(lexer/unicode.c)
        add_expression(lexer/unicode.c → utf8_expr.c)
    endif()

    exports(
        tokenize → public
        lex_init → protected
        lex_state → private
    )
enddeclare()
```

src/tokenizer/lexer/lexer.nlink:

```
# Nested configuration for lexer submodule
inherit_from(../nlink.txt)

configure_lexer()
    buffer_size(8192)

    if(DEBUG_BUILD)
        enable_tracing(ON)
        add_definitions(-DLEXER_DEBUG)
    endif()

    # Local expression overrides
    override_expression(statement.c → stmt_expr.c)
endconfigure()
```

src/cli/nlink.txt:

```
inherit_from(../../pkg.nlink.in)

declare_module(cli)
    scope(public)
    version(1.0.0)
    entry_point(main.c)  # Marks cli/main.c as program entry

    dependencies(
        tokenizer VERSION "^1.0.0"
        parser VERSION "~1.0.0"
        common VERSION ">=0.9.0"
    )

    exports(
```

```
        main → public
        parse_args → protected
        internal_state → private
    )
enddeclare()
```

Inheritance Resolution:

```
1. Start with pkg.nlink.in globals
2. Apply module-level nlink.txt overrides
3. Apply submodule *.nlink local settings
4. Resolve conflicts: Most specific wins
5. Validate no circular dependencies
```

Performance Targets

```toml
[optimization.targets]
size_reduction = "≥40%"        # Minimum binary size reduction
fp_rate = 0.0                  # Zero false positives
tp_rate = 95.0                 # True positive target
build_time = "≤5min"           # Maximum build duration
memory_overhead = "≤100MB"     # Peak memory usage
```

Formal Soundness Proofs

Required Proofs for Each Build:

1. Reachability Proof:

```
For each eliminated symbol S:
  PROVE: ∄ path from entry_points to S

  Where entry_points = {
    cli/main.c:main(),
    ∀ exported symbols in *.nlink,
    ∀ symbols marked scope(public)
  }
```

2. Expression Equivalence Proof:

```
For each transformation T: source.c → expression.c
  PROVE: semantic(source) ≡ semantic(expression)

  Verification:
  - Same input → same output
  - No side effects introduced
  - Complexity O(n) preserved or improved
```

3. Channel Isolation Proof:

```
For channels C₁, C₂ where C₁ ≠ C₂:
  PROVE: symbols(C₁) ∩ symbols(C₂) = ∅

  Specifically:
  - experimental ∩ stable = ∅
  - alpha ∩ production = ∅
```

QA Validation Report Format:

```
{
  "build_id": "2025-01-25-001",
  "quadrant_analysis": {
    "true_positive": 4523,
    "true_negative": 8901,
    "false_positive": 0,
    "false_negative": 127
  },
  "soundness_proofs": {
    "reachability": "VERIFIED",
    "expression_equivalence": "VERIFIED",
    "channel_isolation": "VERIFIED"
  },
  "optimization_metrics": {
    "size_reduction": "47.3%",
    "symbols_eliminated": 4523,
    "symbols_retained": 8901
  },
  "crypto_validation": {
    "shannon_entropy": 7.82,
    "chi_square": 245.7,
    "distribution": "UNIFORM"
  }
}
```

Agent Final Checklist:

```
Before marking build as successful:
□ All FP = 0 (mandatory)
□ All formal proofs verified
□ Size reduction ≥ 40%
□ Crypto modules pass entropy validation
□ No channel cross-contamination
□ All exported symbols preserved
□ Build reproducibility hash generated
□ Signed attestation created
```

🎯 Summary

This document defines the complete waterfall QA process for building `nlink.exe` with:

1. **Feature duplication prevention**:
   - MANDATORY poc/ folder inspection before any implementation
   - Existing code reuse from 7+ POC projects
   - Systematic duplication detection workflow
2. **Quality assurance heuristics**:
   - Every if/else branch analyzed for TP/TN/FP/FN
   - Statement → Expression mappings verified
   - Existing QA patterns from nlink_qa_poc/ reused
3. **Dual-channel architecture**:
   - **Beta channel** for feature development and testing
   - **Alpha channel** for production deployments
   - Schema supports adding custom channels
4. **Whitelist/Blacklist manifest layer**:
   - Like `.gitignore` but for build artifacts
   - Pattern-based inclusion/exclusion
   - Macro and statement-level filtering
5. **CMake-like scripting** for configuration (pkg.nlink.in, *.nlink)
6. **Expression transformation** with duplication checks
7. **State machine minimization** with formal soundness proofs
8. **Zero false-positive** elimination through quadrant analysis
9. **Channel isolation** ensuring Beta features never leak to Alpha builds

Critical Workflow for Every Feature:

```
1. CHECK existing POC implementations
2. ANALYZE if/else branches for QA quadrants
3. REUSE proven patterns from poc/ folders
4. VALIDATE no duplication before coding
5. APPLY TP/TN/FP/FN heuristics to all decisions
```

If/Else Branch QA Mapping:

```
// Every decision point must be analyzed:
if (should_compile_module()) {        // Decision point
    compile_module();                 // TP if correct module compiled
                                      // FN if needed module skipped
} else {
    skip_module();                    // TN if correctly skipped
                                      // FP if needed module skipped
}
// Document: Why is this decision correct?
// Check: Does nlink_qa_poc/ have similar logic?
```

The build system enforces mathematical soundness at every step, ensuring that optimization never compromises correctness, experimental features remain isolated from production code, and NO CODE IS DUPLICATED.

Remember:

- Check poc/ BEFORE coding anything new
- Every if/else needs TP/TN/FP/FN analysis
- Beta is for developers (all features, relaxed checks)
- Alpha is for customers (stable only, strict validation)
- Every elimination must be provably safe
- When in doubt, check existing POC code first

Document Status: Production Ready
Authored by: OBINexus Architecture Team
Methodology: Soundness-First Development with State Machine Minimization
Compliance: NexusLink Standard v1.0.0 (RATIFIED)

🔧 Troubleshooting Common Build Errors

Missing Type Definitions:

If encountering errors like `unknown type name 'etps_context_t'`:

```
// Add to include/nlink/core/types.h:
#include <stdbool.h>
```

```
#include <stdint.h>
#include <stddef.h>

// Forward declarations
typedef struct etps_context etps_context_t;
typedef struct semverx_component semverx_component_t;
typedef struct semverx_range_state semverx_range_state_t;
typedef struct etps_semverx_event etps_semverx_event_t;
typedef struct compatibility_result compatibility_result_t;
typedef struct hotswap_result hotswap_result_t;

// Ensure all headers include base types first:
#include "nlink/core/types.h"
```

Header Include Order:

```
// Correct include order for telemetry.h:
#include <stdbool.h>                    // System headers first
#include <stdint.h>
#include "nlink/core/types.h"           // Base types
#include "nlink/core/etps/etps_types.h" // Module types
#include "nlink/core/etps/telemetry.h"  // Finally, the header
```

Before Adding New Types:

```
# Check if type already exists in POC:
grep -r "typedef.*context_t" nlink*/include/
grep -r "struct.*context" nlink*/src/

# Reuse existing type definitions when possible
```

This ensures all type dependencies are resolved before use AND prevents duplicate type definitions across POC projects.

🔧 Troubleshooting Common Build Errors

Missing Type Definitions:

If encountering errors like `unknown type name 'etps_context_t'`:

```
// Add to include/nlink/core/types.h:
#include <stdbool.h>
#include <stdint.h>
#include <stddef.h>
```

```
// Forward declarations
typedef struct etps_context etps_context_t;
typedef struct semverx_component semverx_component_t;
typedef struct semverx_range_state semverx_range_state_t;
typedef struct etps_semverx_event etps_semverx_event_t;
typedef struct compatibility_result compatibility_result_t;
typedef struct hotswap_result hotswap_result_t;

// Ensure all headers include base types first:
#include "nlink/core/types.h"
```

Header Include Order:

```
// Correct include order for telemetry.h:
#include <stdbool.h>                    // System headers first
#include <stdint.h>
#include "nlink/core/types.h"           // Base types
#include "nlink/core/etps/etps_types.h" // Module types
#include "nlink/core/etps/telemetry.h"  // Finally, the header
```

This ensures all type dependencies are resolved before use.

Glob Pattern Support in nlink CLI

Command Line Invocation with Patterns:

```
# Standard source directory specification
nlink.exe -S . -B build/

# With glob patterns (whitelist/blacklist applied)
nlink.exe -S src/ -B build/ --channel beta

# Pattern-based source selection happens through pkg.nlink.in
```

Whitelist/Blacklist Pattern System:

The `pkg.nlink.in` manifest supports glob wildcards for filtering:

```
configure_whitelist()
    # Source patterns to always process
    pattern("**/*.c")           # All C source files
    pattern("**/*.h")           # All headers
    pattern("**/nlink.txt")     # All module configs
    pattern("**/*.nlink")       # All sub-configs
endconfigure()

configure_blacklist()
```

```
    # Exclude patterns
    pattern("**/tmp/*")         # Temporary directories
    pattern("**/test_*")        # Test files
    pattern("**/*.log")         # Log files
endconfigure()
```

CLI Pattern Processing Flow:

1. **Source Directory (-S)**: Specifies the root directory for pattern matching
2. **Pattern Application**: Whitelist/blacklist patterns filter the source tree
3. **Module Resolution**: The filtered files are processed for build

Pattern Matching in CLI:

The `command_registry.c` shows pattern detection:

```
// Check if the pattern is a glob pattern
if (strchr(pattern, '*') || strchr(pattern, '?')) {
    flags |= NLINK_PATTERN_FLAG_GLOB;
} else {
    // Otherwise use regex
    flags |= NLINK_PATTERN_FLAG_REGEX;
}
```

Entry Point Handling (cli/main.c):

The CLI entry point (`cli/main.c`) is always included as the primary entry point for `nlink.exe`. The build system ensures:

```
declare_module(cli)
    scope(public)
    version(1.0.0)
    entry_point(main.c)  # Marks cli/main.c as program entry
```

Feature Intention with Glob Patterns:

When using glob patterns, the system:

- Applies whitelist patterns first (include matching files)
- Then applies blacklist patterns (exclude specific files)
- Ensures `cli/main.c` remains as the entry point
- Processes expression transformations (e.g., `statement.c → expression.c`)

Example Pattern Reduction:

```
Given: nlink -S src/ with patterns **/*.c
Result after filtering:
- src/cli/main.c          ✓ (entry point, always included)
- src/tokenizer/*.c       ✓ (matches pattern)
- src/parser/*.c          ✓ (matches pattern)
- src/test_*.c            ✗ (blacklisted by **/test_*)
- src/tmp/work.c          ✗ (blacklisted by **/tmp/*)
```