# Contents



# 1 Documentation Technique et Algorithmes Minishell

## 1.1 Table des Matières

7. Algorithmes d'Exécution
8. Cas d'Usage Complexes

---

## 1.2 Algorithmes Principaux

### 1.2.1 1. Algorithme de Tokenisation

**Principe** : Découpe l'input utilisateur en tokens significatifs.

```
// Pseudocode de tokenisation
function tokenize(input, tokens, env, exit_status):
    i = 0
    while i < length(input):
        skip_spaces(input, &i)
        if input[i] == quote:
            handle_quoted_string(input, &i, tokens, env)
        else if is_operator(input[i]):
            parse_operator(input, &i, tokens)
        else:
            parse_word(input, &i, tokens, env, exit_status)
    return tokens
```

**Complexité** : O(n) où n est la longueur de l'input.

**Détails d'implémentation** :

```
int tokenize(const char *input, t_list **tokens, char **env, int exit_status)
{
    char *cleaned;
    t_tokenize_context ctx;
    int result;

    // Nettoyage initial de l'input
    cleaned = input_cleaner((char *)input);
    if (!cleaned)
        return ERROR;

    // Initialisation du contexte
    ctx.env = env;
    ctx.tokens = tokens;
    ctx.exit_status = exit_status;

    // Boucle principale de tokenisation
    result = tokenize_loop_t(cleaned, &ctx);

    free(cleaned);
    return result;
}
```

### 1.2.2 2. Algorithme de Construction d'AST

**Principe** : Construit un arbre syntaxique à partir des tokens selon la priorité des opérateurs.

```
// Pseudocode construction AST
function parse_command(tokens):
    if has_logical_operators(tokens):
        return parse_logical_operators(tokens)
    else if has_pipes(tokens):
        return parse_pipes(tokens)
    else:
        return parse_simple_command(tokens)

function parse_logical_operators(tokens):
    split_point = find_last_logical_operator(tokens)
    left_tokens, right_tokens = split_tokens(tokens, split_point)

    node = create_ast_node(operator_type)
    node.left = parse_command(left_tokens)
    node.right = parse_command(right_tokens)
    return node
```

**Priorité des opérateurs** : 1. Parenthèses `()` (plus haute priorité) 2. Pipes `|` 3. Opérateurs logiques `&&`, `||` 4. Redirections `<`, `>`, `>>`, `<<`

### 1.2.3 3. Algorithme d'Expansion de Variables

**Principe** : Remplace les variables d'environnement par leur valeur.

```
// Pseudocode expansion
function expand_variable(str, pos, env, exit_status):
    if str[pos] == '$':
        if str[pos+1] == '?':
            return itoa(exit_status)
        else:
            var_name = extract_variable_name(str, pos)
            return getenv(env, var_name)
    return null

function remove_quotes_and_expand(input, env):
    result = ""
    i = 0
    while i < length(input):
        if input[i] == single_quote:
            result += handle_single_quotes(input, &i)  // Pas d'expansion
        else if input[i] == double_quote:
            result += handle_double_quotes(input, &i, env)  // Avec expansion
        else if input[i] == '$':
            result += expand_variable(input, &i, env)
        else:
            result += input[i]
            i++
```

```
    return result
```

#### 1.2.4 4. Algorithme d'Exécution d'AST

**Principe** : Parcours post-ordre de l'AST avec gestion des processus.

```
// Pseudocode exécution
function execute_ast(node, user, exiter):
    switch node.type:
        case SIMPLE:
            return execute_simple_command(node, user, exiter)
        case PIPE:
            return execute_pipe(node, user, exiter)
        case AND:
            left_result = execute_ast(node.left, user, exiter)
            if left_result == SUCCESS:
                return execute_ast(node.right, user, exiter)
            return left_result
        case OR:
            left_result = execute_ast(node.left, user, exiter)
            if left_result != SUCCESS:
                return execute_ast(node.right, user, exiter)
            return left_result
```

---

### 1.3 Structures de Données Avancées

#### 1.3.1 1. AST (Abstract Syntax Tree)

**Structure** :

```
        &&
       /  \
    cmd1   ||
          /  \
       cmd2  cmd3
```

**Avantages** : - Représentation naturelle de la priorité des opérateurs - Évaluation récursive simple - Extension facile pour nouveaux opérateurs

**Implémentation** :

```
typedef struct s_ast
{
    t_enum_cmd type;           // Type de nœud
    union {
        struct {               // Pour commandes simples
            char **argv;
            char *infile;
            char *outfile;
            int append_mode;
            char *heredoc_delimiter;
```

```
        } simple;
        struct {                // Pour opérateurs binaires
            struct s_ast *left;
            struct s_ast *right;
        } binary;
    } data;
} t_ast;
```

#### 1.3.2 2. Liste de Tokens

**Structure** : Liste chaînée générique pour flexibilité.

```
typedef struct s_list
{
    void *content;      // Pointeur vers t_token
    struct s_list *next;
} t_list;

typedef struct s_token
{
    int type;           // Type du token
    char *value;        // Valeur string
} t_token;
```

**Avantages** : - Taille dynamique - Insertion/suppression O(1) en tête - Parcours séquentiel efficace

#### 1.3.3 3. Table de Hash pour Environnement (Optimisation Future)

**Principe** : Remplacement du tableau de strings par table de hash.

```
// Structure proposée pour optimisation
typedef struct s_env_entry
{
    char *key;
    char *value;
    struct s_env_entry *next;  // Collision handling
} t_env_entry;

typedef struct s_env_table
{
    t_env_entry **buckets;
    int size;
    int count;
} t_env_table;
```

**Complexité** : - Recherche : O(1) moyen, O(n) pire cas - Insertion : O(1) moyen - Suppression : O(1) moyen

---

## 1.4 Patterns de Conception

### 1.4.1 1. Command Pattern (Builtins)

**Principe** : Encapsulation des commandes builtin dans des structures uniformes.

```c
typedef struct s_builtin
{
    char *name;                     // Nom de la commande
    t_builtin_func func;            // Pointeur vers fonction
} t_builtin;

// Table des builtins
static t_builtin g_builtins[] = {
    {"echo", ms_echo_wrapper},
    {"cd", ms_cd_wrapper},
    {"pwd", ms_pwd_wrapper},
    {"export", ms_env_export_wrapper},
    {"unset", ms_unsetenv_wrapper},
    {"env", ms_env_wrapper},
    {"exit", ms_exit_wrapper},
    {NULL, NULL}
};

// Recherche et exécution
int execute_builtin(const char *cmd, t_user_info *user, t_list *args, int *exiter)
{
    for (int i = 0; g_builtins[i].name; i++) {
        if (ft_strcmp(cmd, g_builtins[i].name) == 0) {
            return g_builtins[i].func(user, args, exiter);
        }
    }
    return NOT_BUILTIN;
}
```

### 1.4.2 2. Visitor Pattern (AST Traversal)

**Principe** : Séparation des algorithmes de traversée de la structure d'arbre.

```c
typedef struct s_ast_visitor
{
    int (*visit_simple)(t_ast *node, void *context);
    int (*visit_pipe)(t_ast *node, void *context);
    int (*visit_and)(t_ast *node, void *context);
    int (*visit_or)(t_ast *node, void *context);
} t_ast_visitor;

int ast_accept(t_ast *node, t_ast_visitor *visitor, void *context)
{
    switch (node->type) {
        case SIMPLE:
            return visitor->visit_simple(node, context);
```

```
        case PIPE:
            return visitor->visit_pipe(node, context);
        case AND:
            return visitor->visit_and(node, context);
        case OR:
            return visitor->visit_or(node, context);
    }
    return ERROR;
}
```

#### 1.4.3 3. Factory Pattern (Token Creation)

**Principe** : Centralisation de la création des tokens.

```
t_token *create_token(int type, const char *value)
{
    t_token *token = malloc(sizeof(t_token));
    if (!token)
        return NULL;

    token->type = type;
    token->value = ft_strdup(value);
    if (!token->value) {
        free(token);
        return NULL;
    }
    return token;
}

t_token *create_operator_token(const char *op)
{
    int type = get_operator_type(op);
    return create_token(type, op);
}

t_token *create_word_token(const char *word)
{
    return create_token(WORD, word);
}
```

#### 1.4.4 4. Strategy Pattern (Redirection Handling)

**Principe** : Différentes stratégies pour différents types de redirections.

```
typedef struct s_redir_strategy
{
    int (*setup)(t_ast *node);
    void (*cleanup)(void);
} t_redir_strategy;

static t_redir_strategy g_redir_strategies[] = {
```

```
    [REDIR_IN] = {setup_input_redirection, cleanup_input_redirection},
    [REDIR_OUT] = {setup_output_redirection, cleanup_output_redirection},
    [REDIR_APPEND] = {setup_append_redirection, cleanup_append_redirection},
    [HEREDOC] = {setup_heredoc, cleanup_heredoc}
};

int setup_redirection(t_ast *node)
{
    t_redir_strategy *strategy = &g_redir_strategies[node->type];
    return strategy->setup(node);
}
```

---

## 1.5 Optimisations

### 1.5.1 1. Optimisation Mémoire

```
#define TOKEN_POOL_SIZE 1000

typedef struct s_token_pool
{
    t_token tokens[TOKEN_POOL_SIZE];
    bool used[TOKEN_POOL_SIZE];
    int next_free;
} t_token_pool;

t_token *alloc_token_from_pool(t_token_pool *pool)
{
    for (int i = pool->next_free; i < TOKEN_POOL_SIZE; i++) {
        if (!pool->used[i]) {
            pool->used[i] = true;
            pool->next_free = i + 1;
            return &pool->tokens[i];
        }
    }
    return malloc(sizeof(t_token));  // Fallback
}
```

#### 1.5.1.1 Pool de Tokens

```
typedef struct s_string_intern
{
    char **strings;
    int count;
    int capacity;
} t_string_intern;

char *intern_string(t_string_intern *intern, const char *str)
```

```
{
    // Recherche si string déjà internée
    for (int i = 0; i < intern->count; i++) {
        if (ft_strcmp(intern->strings[i], str) == 0) {
            return intern->strings[i];  // Retourne référence existante
        }
    }
    // Ajoute nouvelle string
    return add_to_intern(intern, str);
}
```

#### 1.5.1.2 String Interning

### 1.5.2 2. Optimisation Performance

```
typedef struct s_path_cache_entry
{
    char *command;
    char *full_path;
    time_t last_used;
} t_path_cache_entry;

typedef struct s_path_cache
{
    t_path_cache_entry *entries;
    int size;
    int count;
} t_path_cache;

char *cached_resolve_path(t_path_cache *cache, const char *cmd, char **env)
{
    // Recherche dans cache
    for (int i = 0; i < cache->count; i++) {
        if (ft_strcmp(cache->entries[i].command, cmd) == 0) {
            cache->entries[i].last_used = time(NULL);
            return ft_strdup(cache->entries[i].full_path);
        }
    }

    // Résolution et mise en cache
    char *path = resolve_command_path(cmd, env);
    if (path) {
        add_to_cache(cache, cmd, path);
    }
    return path;
}
```

#### 1.5.2.1 Cache de Résolution de Chemins

```c
typedef struct s_wildcard_result
{
    char *pattern;
    char **matches;
    int count;
    bool evaluated;
} t_wildcard_result;

char **get_wildcard_matches(const char *pattern)
{
    static t_wildcard_result cache = {0};

    if (!cache.evaluated || ft_strcmp(cache.pattern, pattern) != 0) {
        free_wildcard_result(&cache);
        cache.matches = expand_wildcards(pattern);
        cache.pattern = ft_strdup(pattern);
        cache.evaluated = true;
    }
    return cache.matches;
}
```

#### 1.5.2.2 Lazy Evaluation pour Wildcards

---

## 1.6 Gestion Mémoire

### 1.6.1 1. Stratégies d'Allocation

```c
typedef struct s_stack_allocator
{
    char *memory;
    size_t size;
    size_t used;
} t_stack_allocator;

void *stack_alloc(t_stack_allocator *allocator, size_t size)
{
    if (allocator->used + size > allocator->size) {
        return NULL;  // Stack overflow
    }
    void *ptr = allocator->memory + allocator->used;
    allocator->used += size;
    return ptr;
}

void stack_reset(t_stack_allocator *allocator)
{
    allocator->used = 0;  // Reset complet
```

```
}
```

#### 1.6.1.1 Stack Allocator pour Parsing Temporaire

```
typedef struct s_ref_string
{
    char *data;
    int ref_count;
} t_ref_string;

t_ref_string *create_ref_string(const char *str)
{
    t_ref_string *ref_str = malloc(sizeof(t_ref_string));
    ref_str->data = ft_strdup(str);
    ref_str->ref_count = 1;
    return ref_str;
}

void retain_string(t_ref_string *ref_str)
{
    ref_str->ref_count++;
}

void release_string(t_ref_string *ref_str)
{
    ref_str->ref_count--;
    if (ref_str->ref_count == 0) {
        free(ref_str->data);
        free(ref_str);
    }
}
```

#### 1.6.1.2 Reference Counting pour Strings

### 1.6.2 2. Détection de Fuites

```
#ifdef DEBUG_MEMORY
typedef struct s_alloc_info
{
    void *ptr;
    size_t size;
    const char *file;
    int line;
    struct s_alloc_info *next;
} t_alloc_info;

static t_alloc_info *g_allocations = NULL;
```

```c
void *debug_malloc(size_t size, const char *file, int line)
{
    void *ptr = malloc(size);
    if (ptr) {
        t_alloc_info *info = malloc(sizeof(t_alloc_info));
        info->ptr = ptr;
        info->size = size;
        info->file = file;
        info->line = line;
        info->next = g_allocations;
        g_allocations = info;
    }
    return ptr;
}

#define malloc(size) debug_malloc(size, __FILE__, __LINE__)
#endif
```

#### 1.6.2.1 Memory Tracking

---

## 1.7 Algorithmes de Parsing

### 1.7.1 1. Recursive Descent Parser

**Principe** : Parser récursif descendant pour la grammaire du shell.

```
Grammar:
command_line    → logical_expr
logical_expr    → pipe_expr (('&&' | '||') pipe_expr)*
pipe_expr       → simple_cmd ('|' simple_cmd)*
simple_cmd      → WORD*
```

**Implémentation** :

```c
t_ast *parse_command_line(t_list **tokens)
{
    return parse_logical_expr(tokens);
}

t_ast *parse_logical_expr(t_list **tokens)
{
    t_ast *left = parse_pipe_expr(tokens);

    while (*tokens && is_logical_operator((*tokens)->content)) {
        t_token *op = (*tokens)->content;
        *tokens = (*tokens)->next;

        t_ast *right = parse_pipe_expr(tokens);

        t_ast *node = new_ast_node(get_operator_type(op->value));
```

```
        node->left = left;
        node->right = right;
        left = node;
    }
    return left;
}
```

### 1.7.2 2. Shunting Yard Algorithm (Alternative)

**Principe** : Conversion de notation infixe vers notation postfixe pour évaluation.

```
// Pseudocode Shunting Yard
function shunting_yard(tokens):
    output_queue = []
    operator_stack = []

    for each token in tokens:
        if token is operand:
            output_queue.push(token)
        else if token is operator:
            while (operator_stack.top() has higher precedence):
                output_queue.push(operator_stack.pop())
            operator_stack.push(token)
        else if token is left_paren:
            operator_stack.push(token)
        else if token is right_paren:
            while operator_stack.top() != left_paren:
                output_queue.push(operator_stack.pop())
            operator_stack.pop()  // Remove left_paren

    while operator_stack is not empty:
        output_queue.push(operator_stack.pop())

    return output_queue
```

### 1.7.3 3. Quote Handling Algorithm

**Principe** : Machine à états pour gérer les différents types de quotes.

```
typedef enum e_quote_state
{
    NORMAL,
    IN_SINGLE_QUOTE,
    IN_DOUBLE_QUOTE,
    ESCAPED
} t_quote_state;

int parse_with_quotes(const char *input, char **result)
{
    t_quote_state state = NORMAL;
    int i = 0, j = 0;
```

```
    char *output = malloc(ft_strlen(input) + 1);

    while (input[i]) {
        switch (state) {
            case NORMAL:
                if (input[i] == '\'') {
                    state = IN_SINGLE_QUOTE;
                } else if (input[i] == '"') {
                    state = IN_DOUBLE_QUOTE;
                } else if (input[i] == '\\') {
                    state = ESCAPED;
                } else {
                    output[j++] = input[i];
                }
                break;
            case IN_SINGLE_QUOTE:
                if (input[i] == '\'') {
                    state = NORMAL;
                } else {
                    output[j++] = input[i];
                }
                break;
            case IN_DOUBLE_QUOTE:
                if (input[i] == '"') {
                    state = NORMAL;
                } else if (input[i] == '$') {
                    // Expansion dans double quotes
                    i += expand_variable(&input[i], &output[j]);
                    continue;
                } else {
                    output[j++] = input[i];
                }
                break;
            case ESCAPED:
                output[j++] = input[i];
                state = NORMAL;
                break;
        }
        i++;
    }
    output[j] = '\0';
    *result = output;
    return (state == NORMAL) ? SUCCESS : ERROR;
}
```

---

## 1.8 Algorithmes d'Exécution

### 1.8.1 1. Pipeline Execution

**Principe** : Création de pipes et fork de processus en chaîne.

```c
int execute_pipeline(t_ast *node, t_user_info *user, int *exiter)
{
    int pfd[2];
    pid_t pid_left, pid_right;
    int status = 0;

    // Création du pipe
    if (pipe(pfd) == -1) {
        perror("pipe");
        return ERROR;
    }

    // Fork pour processus gauche
    pid_left = fork();
    if (pid_left == 0) {
        // Processus enfant gauche
        close(pfd[0]);              // Ferme lecture
        dup2(pfd[1], STDOUT_FILENO); // Redirige stdout vers pipe
        close(pfd[1]);
        exit(execute_ast(node->left, user, exiter));
    }

    // Fork pour processus droit
    pid_right = fork();
    if (pid_right == 0) {
        // Processus enfant droit
        close(pfd[1]);              // Ferme écriture
        dup2(pfd[0], STDIN_FILENO); // Redirige stdin depuis pipe
        close(pfd[0]);
        exit(execute_ast(node->right, user, exiter));
    }

    // Processus parent
    close(pfd[0]);
    close(pfd[1]);

    // Attente des processus enfants
    waitpid(pid_left, NULL, 0);
    waitpid(pid_right, &status, 0);

    return WEXITSTATUS(status);
}
```

### 1.8.2 2. Redirection Handling

**Principe** : Sauvegarde et restauration des descripteurs de fichiers.

```c
typedef struct s_redir_backup
{
    int stdin_backup;
    int stdout_backup;
    int stderr_backup;
} t_redir_backup;

int setup_redirections(t_ast *node, t_redir_backup *backup)
{
    // Sauvegarde des descripteurs originaux
    backup->stdin_backup = dup(STDIN_FILENO);
    backup->stdout_backup = dup(STDOUT_FILENO);
    backup->stderr_backup = dup(STDERR_FILENO);

    // Redirection d'entrée
    if (node->infile) {
        int fd = open(node->infile, O_RDONLY);
        if (fd == -1) {
            perror(node->infile);
            return ERROR;
        }
        dup2(fd, STDIN_FILENO);
        close(fd);
    }

    // Redirection de sortie
    if (node->outfile) {
        int flags = O_WRONLY | O_CREAT;
        if (node->append_mode) {
            flags |= O_APPEND;
        } else {
            flags |= O_TRUNC;
        }

        int fd = open(node->outfile, flags, 0644);
        if (fd == -1) {
            perror(node->outfile);
            return ERROR;
        }
        dup2(fd, STDOUT_FILENO);
        close(fd);
    }

    return SUCCESS;
}

void restore_redirections(t_redir_backup *backup)
{
    dup2(backup->stdin_backup, STDIN_FILENO);
```

```
    dup2(backup->stdout_backup, STDOUT_FILENO);
    dup2(backup->stderr_backup, STDERR_FILENO);

    close(backup->stdin_backup);
    close(backup->stdout_backup);
    close(backup->stderr_backup);
}
```

#### 1.8.3 3. Heredoc Implementation

**Principe** : Lecture interactive jusqu'au délimiteur avec pipe temporaire.

```
int handle_heredoc(const char *delimiter, char **env)
{
    int pfd[2];
    char *line;

    if (pipe(pfd) == -1) {
        perror("pipe");
        return -1;
    }

    // Lecture des lignes jusqu'au délimiteur
    while ((line = readline("> ")) != NULL) {
        if (ft_strcmp(line, delimiter) == 0) {
            free(line);
            break;
        }

        // Expansion des variables dans heredoc
        char *expanded = remove_quotes_and_expand(line, env);
        write(pfd[1], expanded, ft_strlen(expanded));
        write(pfd[1], "\n", 1);

        free(line);
        free(expanded);
    }

    close(pfd[1]);
    return pfd[0];  // Retourne fd de lecture
}
```

---

### 1.9 Cas d'Usage Complexes

#### 1.9.1 1. Commande avec Multiples Redirections

```
command < input.txt > output.txt 2>>error.log
```

**Algorithme** :

```c
int handle_multiple_redirections(t_ast *node)
{
    // Traitement dans l'ordre : < puis > puis 2>>
    if (node->infile) {
        setup_input_redirection(node->infile);
    }
    if (node->outfile) {
        if (node->append_mode) {
            setup_append_redirection(node->outfile);
        } else {
            setup_output_redirection(node->outfile);
        }
    }
    if (node->stderr_to_devnull) {
        int devnull = open("/dev/null", O_WRONLY);
        dup2(devnull, STDERR_FILENO);
        close(devnull);
    }
    return SUCCESS;
}
```

### 1.9.2 2. Pipeline avec Opérateurs Logiques

```
cmd1 | cmd2 && cmd3 || cmd4
```

**Structure AST** :

```
        ||
       /  \
     &&    cmd4
    /  \
   |    cmd3
  / \
cmd1 cmd2
```

**Exécution** :

```c
int execute_complex_command(t_ast *node)
{
    // Évaluation court-circuit
    switch (node->type) {
        case OR:
            if (execute_ast(node->left, user, exiter) == SUCCESS) {
                return SUCCESS;  // Court-circuit : pas besoin d'évaluer droite
            }
            return execute_ast(node->right, user, exiter);
        case AND:
            if (execute_ast(node->left, user, exiter) != SUCCESS) {
                return ERROR;    // Court-circuit : pas besoin d'évaluer droite
            }
            return execute_ast(node->right, user, exiter);
```

```
    }
}
```

### 1.9.3 3. Expansion Complexe de Variables

```
echo "${HOME}/bin:${PATH}" > config
```

**Algorithme d'expansion** :

```
char *expand_complex_variable(const char *input, char **env)
{
    char *result = ft_strdup("");
    int i = 0;

    while (input[i]) {
        if (input[i] == '$' && input[i+1] == '{') {
            // Variable avec accolades
            int start = i + 2;
            int end = find_closing_brace(input, start);
            char *var_name = ft_substr(input, start, end - start);
            char *var_value = ms_getenv(env, var_name, ft_strlen(var_name));

            result = ft_strjoin_free(result, var_value ? var_value : "");
            free(var_name);
            i = end + 1;
        } else if (input[i] == '$') {
            // Variable simple
            char *var_name = extract_simple_var_name(input, &i);
            char *var_value = ms_getenv(env, var_name, ft_strlen(var_name));

            result = ft_strjoin_free(result, var_value ? var_value : "");
            free(var_name);
        } else {
            result = append_char(result, input[i]);
            i++;
        }
    }
    return result;
}
```

### 1.9.4 4. Gestion des Signaux pendant l'Exécution

```
volatile sig_atomic_t g_received_signal = 0;

void signal_handler(int sig)
{
    g_received_signal = sig;
}

int execute_with_signal_handling(t_ast *node, t_user_info *user)
```

```
{
    struct sigaction sa;
    sa.sa_handler = signal_handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;

    sigaction(SIGINT, &sa, NULL);
    sigaction(SIGQUIT, &sa, NULL);

    int result = execute_ast(node, user, NULL);

    if (g_received_signal) {
        // Traitement du signal reçu
        handle_received_signal(g_received_signal);
        g_received_signal = 0;
    }

    return result;
}
```

---

## 1.10 Conclusion

Cette documentation technique couvre les aspects algorithmiques et d'implémentation avancés du projet Minishell. Les algorithmes choisis privilégient la clarté et la maintenabilité tout en conservant des performances acceptables pour un shell interactif.

### 1.10.1 Points Clés Techniques

1. **Parsing Récursif** : Simple à implémenter et à maintenir
2. **AST** : Représentation naturelle des commandes complexes
3. **Gestion Mémoire** : Stratégies d'optimisation pour performance
4. **Patterns de Conception** : Code modulaire et extensible
5. **Algorithmes Éprouvés** : Basés sur les standards Unix

Cette architecture permet une extension facile pour de nouvelles fonctionnalités tout en maintenant la robustesse du système.