# Core Skills

- /build-plan
- /plan-with-team
- /learning-opportunity
- /code-review
- /security-audit

# /build-plan

**Source:** `~/.claude/skills/build-plan/SKILL.md`

---

name: build-plan version: 2.0 description: > Execute an approved plan using TaskList with builder/validator teams. Use after /plan-with-team creates an approved plan. Triggers: "execute the plan", "build this", "implement the plan", "run the plan", "/build-plan". argument-hint: "[path to plan file, or leave empty for latest in ~/system/specs/]" level: 3

# /build-plan — Execute Approved Plan

Purpose

Executes a pre-approved plan file using parallel builder/validator TaskList agents. Run AFTER `/plan-with-team` — never execute without an approved plan.

# Variables

| Variable | Type | Description | Default |
|---|---|---|---|
| `plan_path` | path | Path to plan file | latest in ~/system/specs/ |
| `concurrency` | number | Parallel builders | 3 |
| `yolo` | flag | Skip browser testing | false |

# Workflow

## Step 1: Load Plan

- if `$ARGUMENTS` provided → use as plan_path
- else → find latest plan: `ls -t ~/system/specs/*.md | head -1`
- Read plan file, verify it has: [ ] task checkboxes + acceptance criteria

## Step 2: Validate Prerequisites

- Check: does plan have approval marker? (`## APPROVED` section or `status: approved`)
- if not approved → STOP, tell user to run `/plan-with-team` first
- Check: are required services running? (DB, relevant tools)

## Step 3: Activate Build Mode

```
node ~/system/tools/build-mode.js start <project_dir> --concurrency <N>
```

## Builder completion flow

Builders call `node ~/system/tools/mc.js ready <id>` NOT `mc.js done`. Only after Proveo/validator verification can tasks be marked done. Build-plan CANNOT report COMPLETE until all tasks pass Proveo verification.

# Step 4: Execute Tasks (parallel where independent)

For each `[ ]` task in plan:

- if task has no dependencies → spawn in parallel with builder agent
- if task has dependencies → wait for dep completion, then spawn
- Each builder: reads task spec, implements, marks `[x]` when done

# Step 5: Validate Each Task

After each task completes → spawn validator agent:

- Runs tests relevant to task
- Checks acceptance criteria
- if FAIL → send back to builder with failure context
- if PASS → proceed

# Step 6: Final Check

- All tasks `[x]`? → run full test suite
- if PASS → `node ~/system/tools/build-mode.js stop --status completed`
- if FAIL → `node ~/system/tools/build-mode.js stop --status failed` + report

# Report Format

```
BUILD-PLAN EXECUTION REPORT
Plan: [filename]
Status: [COMPLETE|PARTIAL|FAILED]
Tasks: X/Y completed
Tests: [passing/total]
Duration: [time]
Issues: [any blockers or failures]
```

$ARGUMENTS

# /plan-with-team

## **Source:** `~/.claude/skills/plan-with-team/SKILL.md`

---

# name: plan-with-team description: Create implementation plans with builder/validator agent teams. Use for major features, refactoring, or system components. argument-hint: "[what to build]"

# Plan With Team

Create an implementation plan with builder/validator agent teams.

# Related Skills (pick the right one)

- `/plan-with-team` (this skill) — Multi-expert planning for complex features, refactors, system components.
- `/build-plan` — Execute the plan produced here. Run AFTER plan is approved.
- `/hop-build <task_id>` — For a SINGLE task (not a multi-task plan).
- `/build` — Toggle session into Build Mode.
- `/prime-build` — Load lightweight build context into session.

# Instructions

You are creating a detailed implementation plan that assigns work to specialized agent teams.

# Step 1: Research (MANDATORY)

Before planning, research:

1. **If building a company/org:** Find 2-3 existing examples, analyze their structure
2. **If building software:** Explore the codebase, understand existing patterns
3. **If building process:** Find industry standards and best practices

Use Glob, Grep, Read, WebSearch as needed. Document findings.

# Step 2: Analyze

Based on research:

- What patterns should we copy?
- What should we adapt for our needs?
- What are the key components?

# Step 3: Define Team

For each major task, assign:

- **Builder** — Creates/implements
- **Validator** — Verifies the work

Reference agents from `~/.claude/agents/`:

- `builder.md` — Implementation agent
- `validator.md` — Verification agent (read-only)

# Step 4: Create Plan

Write plan to `~/system/specs/<name>-plan.md` with this structure:

```
# Plan: [Name]


## Research Summary
```

```
[What we learned from existing examples]

## Objective
[1-2 sentences]

## Team Orchestration

### Team Members
| ID | Name | Role | Agent Type |
|----|------|------|------------|
| B1 | [name]-builder | Build [what] | builder |
| V1 | [name]-validator | Validate [what] | validator |

### Step-by-Step Tasks

#### Phase 1: [Name]
**Task 1:** [Description]
- Owner: B1
- BlockedBy: none
- Acceptance:
  - [ ] Criterion 1
  - [ ] Criterion 2

**Task 2:** Validate [above]
- Owner: V1
- BlockedBy: 1
- Acceptance: [criteria]

## Validation Commands
[How to verify the work]
```

# Step 5: Validate Plan

Self-validate the plan:

- All tasks have acceptance criteria
- Dependencies make sense (validators blocked by builders)
- No circular dependencies

**MANDATORY CHECK — plan is INCOMPLETE without both:**

☐ **Validation task** exists — owner: Proveo/Angie Jones, end-to-end test with real evidence (not dry-run), BlockedBy all build tasks

☐ **Documentation task** exists — owner: Skillforge, BookStack page for every system built or changed, BlockedBy validation task

If either is missing → add them before presenting to CEO. Do not ask. Just add.

# Step 6: Present for Approval

Show the user:

1. Research summary
2. Plan overview
3. Number of tasks and phases
4. Ask: "Approve plan? Then run `/build-plan` to execute."

# Output

The plan file path: `~/system/specs/<name>-plan.md`

Ready for execution with `/build-plan`.

$ARGUMENTS

# /learning-opportunity

**Source:** `~/.claude/skills/learning-opportunity/SKILL.md`

---

name: learning-opportunity description: Self-improving feedback loop. When something goes wrong, analyze root cause, patch the system, and ensure it never happens again. argument-hint: "[describe what went wrong]"

# Learning Opportunity

Turn every mistake into a permanent system improvement.

## Instructions

You are analyzing a mistake or failure and patching the system so it never recurs.

**Principle:** AI bez enforcement-a ne radi. Markdown rules = suggestions. Hooks/scripts = enforcement. Always prefer deterministic fixes over documentation fixes.

### Step 1: Identify the Failure

If argument provided, use it. Otherwise, ask:

- What went wrong?
- When did it happen?
- What was the expected vs actual outcome?

Classify the failure type:

- **HALLUCINATION** — AI invented something that doesn't exist (tool, path, port, import)
- **PROCESS_SKIP** — AI skipped a required step (no boot, no backup, no task)
- **WRONG_OUTPUT** — AI produced incorrect content (wrong data, bad code, broken logic)
- **KNOWLEDGE_GAP** — AI didn't know something it should have known
- **REPEAT_MISTAKE** — Same error as a previous session (worst category)

# Step 2: Root Cause Analysis

Trace the failure through GOTCHA layers:

1. **Goals** — Was there a spec/rule that should have prevented this?

```
# Check existing rules
ls ~/system/rules/
grep -r "relevant keyword" ~/system/rules/
```

2. **Tools** — Did a tool fail or was a phantom tool used?

```
# Check manifest
grep "relevant tool" ~/system/tools/manifest.md
```

3. **Context** — Was the context missing or wrong?

```
# Check HiveMind for prior knowledge
node ~/system/agents/hivemind/hivemind.js query "relevant keyword"
```

4. **Hooks** — Should an enforcement hook have caught this?

```
# Check existing hooks
ls ~/.claude/hooks/
```

5. **Memory** — Was this a known issue that was forgotten?

```
# Check memory files
grep "relevant keyword" ~/.claude/projects/-Users-makinja/memory/MEMORY.md
```

Document: Which layer failed? Why?

# Step 3: Determine Fix Type

Choose the STRONGEST fix available (top = strongest):

| Priority | Fix Type | When to Use |
|---|---|---|
| 1 | **Hook** (Python enforcement) | Hallucinations, phantom tools, security violations |
| 2 | **Tool update** (deterministic code) | Missing validation, wrong behavior |
| 3 | **Rule addition** (~/system/rules/) | New process requirement, agent behavior |
| 4 | **CLAUDE.md update** | Missing instruction, wrong priority |
| 5 | **Memory update** | Lesson learned, context for future |

**NEVER** use only option 5 alone. Memory without enforcement = ZAKON #1 violation.

# Step 4: Apply the Patch

Based on fix type, apply changes:

## If HALLUCINATION ? Update hallucination-detector.py

```
# Read current blocklist
grep -A 50 "PHANTOM_TOOLS" ~/.claude/hooks/hallucination-detector.py
```

Add the hallucinated item to the appropriate blocklist (PHANTOM_TOOLS, KNOWN_PORTS, etc.)

## If PROCESS_SKIP ? Update/create enforcement hook

Check if gotcha-enforcer.py can be extended, or create new hook.

## If WRONG_OUTPUT ? Update tool or add validation

Fix the tool that produced wrong output. Add input validation.

## If KNOWLEDGE_GAP ? Add to context + memory

```
# Add to HiveMind
node ~/system/agents/hivemind/hivemind.js post john lesson "description"
```

## If REPEAT_MISTAKE ? Escalate enforcement

If this mistake happened before, the previous fix was too weak. Go UP the priority list (e.g., if rule exists but wasn't followed → add hook).

# Step 5: Verify the Fix

Test that the fix actually works:

- If hook: test with a simulated bad input
- If tool: run the tool and verify output
- If rule: check it's in the right location and formatted correctly

# Step 6: Log Everything

```
# 1. Log to CHANGELOG
bash ~/system/tools/syslog.sh add "LEARNING: [description] — fix: [what was changed]"

# 2. Log to HiveMind
node ~/system/agents/hivemind/hivemind.js post john lesson "[failure type]: [what happened] →
[what was fixed]"

# 3. Update lessons-learned if exists
# ~/system/rules/lessons-learned.md
```

# Step 7: Report

Show the user:

```
## Learning Opportunity Report

### Failure
- **Type:** [HALLUCINATION|PROCESS_SKIP|WRONG_OUTPUT|KNOWLEDGE_GAP|REPEAT_MISTAKE]
- **Description:** [what went wrong]
- **Root Cause:** [which GOTCHA layer failed and why]

### Fix Applied
- **Fix Type:** [Hook|Tool|Rule|CLAUDE.md|Memory]
- **File Changed:** [path]
- **What Changed:** [description]

### Enforcement Level
- [ ] Deterministic (hook/script blocks bad behavior)
- [ ] Documented (rule/instruction guides good behavior)
- [ ] Remembered (memory/HiveMind for context)
```

```
### Verification
- [ ] Fix tested and working
- [ ] Logged to CHANGELOG
- [ ] Logged to HiveMind
```

# Rules

1. **Deterministic > Documented** — A hook that blocks is worth 100 markdown rules
2. **ZAKON #1 applies** — If the fix is "write more markdown", it's NOT a fix
3. **Escalate repeats** — Same mistake twice = previous fix was too weak
4. **Always log** — CHANGELOG + HiveMind, no exceptions
5. **Backup first** — `setup-backup.sh` before any hook/tool changes

$ARGUMENTS

# /code-review

**Source:** `~/.claude/skills/code-review/SKILL.md`

---

# name: code-review version: "2.0" level: 3 trigger: "code review, review this code, check my code, pre-commit review, security review of code" author: john updated: 2026-03-16

# Code Review — Level 3 Supervised Skill

Sistematski code review sa if/then control flow. Security-first, actionable feedback.

## WHEN TO USE

- IF "review", "code review", "check code", "pre-commit" → activate this skill
- IF security-specific request → prioritize Security section, run sentry-security-review first

## WORKFLOW

### Step 1: Scope Check

```
IF large PR (>500 lines):
  → Split: delegate security to securion sub-agent, delegate logic to code-reviewer sub-agent
  → Merge reports before final output
ELSE:
  → Single-pass review, continue to Step 2
```

# Step 2: RAG Context

```
node ~/system/tools/rag-router.js query "code review patterns [tech stack]" --top 3
```

Check HiveMind for prior decisions on this codebase.

# Step 3: Security Scan (ALWAYS FIRST)

```
IF bash/shell code detected → check for injection patterns
IF database queries → check for SQL injection
IF user input handling → check XSS, validation
IF credentials/keys visible → STOP, report immediately (Level 5 block)
```

# Step 4: Quality Checklist

```
[ ] Correctness: edge cases, error handling, null safety
[ ] Security: OWASP Top 10, no hardcoded secrets, input validation
[ ] Performance: N+1 queries, unnecessary loops, memory leaks
[ ] Maintainability: DRY, naming, dead code
[ ] Tests: coverage for happy path + 2 edge cases minimum
[ ] GOTCHA: does this introduce regressions?
```

# Step 5: Report Format

```
IF critical security issue → status: BLOCKED, stop review, escalate
IF blocking bugs (>3) → status: NEEDS_WORK
IF minor issues only → status: APPROVED_WITH_COMMENTS
IF clean → status: APPROVED
```

# OUTPUT FORMAT (report to John, not user)

```
CODE REVIEW REPORT

Status: APPROVED | APPROVED_WITH_COMMENTS | NEEDS_WORK | BLOCKED
Files reviewed: [list]
Lines reviewed: [count]

□□CRITICAL (must fix before merge):
  - [issue]: [file:line] — [fix suggestion]

□□SHOULD FIX:
  - [issue]: [file:line] — [suggestion]

□□OPTIONAL:
  - [suggestion]

Security: PASS | WARN | FAIL
Tests: [coverage%] | [missing]
Verdict: [one sentence summary]
```

# ESCALATION

- Security FAIL → delegate to securion agent immediately
- Architectural concern → delegate to sentinel-architect agent
- Performance concern → query HiveMind for prior benchmarks first

# /security-audit

**Source:** `~/.claude/skills/security-audit/SKILL.md`

---

name: security-audit version: 2.0 description: > Run comprehensive security audit following OWASP and ALAI LAWS. Use for: "security review", "audit this code", "check for vulnerabilities", "OWASP check", "before deploying", "security scan", "/security-audit". level: 3 company: Securion

# /security-audit — Security Review

## Purpose

Systematic security review covering OWASP Top 10, ALAI internal LAWS, and code-specific vulnerabilities.

# Variables

| Variable | Type | Description | Default |
|---|---|---|---|
| `target` | path/scope | File, directory, or "full system" | current project |
| `depth` | string | quick \| standard \| deep | standard |
| `focus` | string | owasp \| laws \| auth \| api \| all | all |

# Workflow

## Step 1: Scope

- Read $ARGUMENTS to determine target and depth
- if no target → audit current working directory
- if depth=quick → run only LAWS + auth checks
- if depth=deep → run all + tob-* skill checks

## Step 2: ALAI LAWS Compliance

Check each LAW:

- **ZAKON 0 (Tajnost)**: No secrets in code, no internal URLs exposed, no employee data hardcoded
- **ZAKON 1 (Ne škodi)**: No destructive ops without confirm, backups exist for critical data
- **ZAKON 2 (Slušaj)**: Auth on all endpoints, RBAC, admin routes protected
- **ZAKON 3 (Čuvaj sebe)**: Error handling, graceful degradation, no crash on bad input

## Step 3: OWASP Top 10 Check

For each category, scan target:

1. Injection (SQL, NoSQL, command injection)
2. Broken Authentication (weak JWT, no rate limit, session issues)
3. Sensitive Data Exposure (logs, responses, hardcoded secrets)
4. Security Misconfiguration (CORS, headers, default credentials)

5. XSS (reflected, stored, DOM-based)
6. Broken Access Control (IDOR, privilege escalation)
7. Vulnerable Dependencies (`npm audit` or equivalent)
8. Insecure Deserialization
9. Logging & Monitoring gaps
10. SSRF

# Step 4: Run Available Tools

- if tob-static-analysis available → run on target
- if tob-insecure-defaults available → check configs
- if tob-sharp-edges available → check dangerous patterns
- `npm audit --audit-level=high` if package.json exists

# Step 5: Report

- if CRITICAL found → flag for immediate fix, offer to create MC task
- if HIGH found → list with recommended fixes
- if depth=deep → include code snippets for each finding

# Report Format

```
SECURITY AUDIT REPORT
Target: [scope]
Depth: [quick|standard|deep]
Date: [timestamp]

CRITICAL (block deployment):
  [C1] [finding] — [file:line] — [fix]

HIGH (fix before next release):
  [H1] [finding] — [fix]

MEDIUM:
  [M1] [finding]

LAWS: [PASS|FAIL — list failures]
OWASP: [X/10 categories clean]
Tools run: [list]
```

$ARGUMENTS