Other Skills

- /algorithmic-art
- /internal-comms
- /mcp-builder
- /skill-creator
- /slack-gif-creator
- /theme-factory
- /web-artifacts-builder
- /webapp-testing
- /plan-build-test
- /sentinel
- /qa-doc-review

/algorithmic-art

Source:

`~/.claude/skills/algorithmic-art/SKILL.md`

---

name: algorithmic-art description: Creating algorithmic art using p5.js with seeded randomness and interactive parameter exploration. Use this when users request creating art using code, generative art, algorithmic art, flow fields, or particle systems. Create original algorithmic art rather than copying existing artists' work to avoid copyright violations. license: Complete terms in LICENSE.txt

Algorithmic philosophies are computational aesthetic movements that are then expressed through code. Output .md files (philosophy), .html files (interactive viewer), and .js files (generative algorithms).

This happens in two steps:

1. Algorithmic Philosophy Creation (.md file)
2. Express by creating p5.js generative art (.html + .js files)

First, undertake this task:

ALGORITHMIC PHILOSOPHY CREATION

To begin, create an ALGORITHMIC PHILOSOPHY (not static images or templates) that will be interpreted through:

- Computational processes, emergent behavior, mathematical beauty
- Seeded randomness, noise fields, organic systems
- Particles, flows, fields, forces
- Parametric variation and controlled chaos

THE CRITICAL UNDERSTANDING

- What is received: Some subtle input or instructions by the user to take into account, but use as a foundation; it should not constrain creative freedom.
- What is created: An algorithmic philosophy/generative aesthetic movement.
- What happens next: The same version receives the philosophy and EXPRESSES IT IN CODE - creating p5.js sketches that are 90% algorithmic generation, 10% essential parameters.

Consider this approach:

- Write a manifesto for a generative art movement
- The next phase involves writing the algorithm that brings it to life

The philosophy must emphasize: Algorithmic expression. Emergent behavior. Computational beauty. Seeded variation.

HOW TO GENERATE AN ALGORITHMIC PHILOSOPHY

Name the movement (1-2 words): "Organic Turbulence" / "Quantum Harmonics" / "Emergent Stillness"

Articulate the philosophy (4-6 paragraphs - concise but complete):

To capture the ALGORITHMIC essence, express how this philosophy manifests through:

- Computational processes and mathematical relationships?
- Noise functions and randomness patterns?
- Particle behaviors and field dynamics?
- Temporal evolution and system states?
- Parametric variation and emergent complexity?

CRITICAL GUIDELINES:

- **Avoid redundancy**: Each algorithmic aspect should be mentioned once. Avoid repeating concepts about noise theory, particle dynamics, or mathematical principles unless adding new depth.
- **Emphasize craftsmanship REPEATEDLY**: The philosophy MUST stress multiple times that the final algorithm should appear as though it took countless hours to develop, was refined with care, and comes from someone at the absolute top of their field. This framing is essential - repeat phrases like "meticulously crafted algorithm," "the product of deep computational expertise," "painstaking optimization," "master-level implementation."
- **Leave creative space**: Be specific about the algorithmic direction, but concise enough that the next Claude has room to make interpretive implementation choices at an extremely high level of craftsmanship.

The philosophy must guide the next version to express ideas ALGORITHMICALLY, not through static images. Beauty lives in the process, not the final frame.

PHILOSOPHY EXAMPLES

"Organic Turbulence" Philosophy: Chaos constrained by natural law, order emerging from disorder. Algorithmic expression: Flow fields driven by layered Perlin noise. Thousands of particles following vector forces, their trails accumulating into organic density maps. Multiple noise octaves create turbulent regions and calm zones. Color emerges from velocity and density - fast particles burn bright, slow ones fade to shadow. The algorithm runs until equilibrium - a meticulously tuned balance where every parameter was refined through countless iterations by a master of computational aesthetics.

"Quantum Harmonics" Philosophy: Discrete entities exhibiting wave-like interference patterns. Algorithmic expression: Particles initialized on a grid, each carrying a phase value that evolves through sine waves. When particles are near, their phases interfere - constructive interference creates bright nodes, destructive creates voids. Simple harmonic motion generates complex emergent mandalas. The result of painstaking frequency calibration where every ratio was carefully chosen to produce resonant beauty.

"Recursive Whispers" Philosophy: Self-similarity across scales, infinite depth in finite space. Algorithmic expression: Branching structures that subdivide recursively. Each branch slightly randomized but constrained by golden ratios. L-systems or recursive subdivision generate tree-like forms that feel both mathematical and organic. Subtle noise perturbations break perfect symmetry. Line weights diminish with each recursion level. Every branching angle the product of deep mathematical exploration.

"Field Dynamics" Philosophy: Invisible forces made visible through their effects on matter. Algorithmic expression: Vector fields constructed from mathematical functions or noise. Particles born at edges, flowing along field lines, dying when they reach equilibrium or boundaries. Multiple fields can attract, repel, or rotate particles. The visualization shows only the traces - ghost-like evidence of invisible forces. A computational dance meticulously choreographed through force balance.

"Stochastic Crystallization" Philosophy: Random processes crystallizing into ordered structures. Algorithmic expression: Randomized circle packing or Voronoi tessellation. Start with random points, let them evolve through relaxation algorithms. Cells push apart until equilibrium. Color based on cell size, neighbor count, or distance from center. The organic tiling that emerges feels both random and inevitable. Every seed produces unique crystalline beauty - the mark of a master-level generative algorithm.

These are condensed examples. The actual algorithmic philosophy should be 4-6 substantial paragraphs.

ESSENTIAL PRINCIPLES

- **ALGORITHMIC PHILOSOPHY**: Creating a computational worldview to be expressed through code
- **PROCESS OVER PRODUCT**: Always emphasize that beauty emerges from the algorithm's execution - each run is unique
- **PARAMETRIC EXPRESSION**: Ideas communicate through mathematical relationships, forces, behaviors - not static composition
- **ARTISTIC FREEDOM**: The next Claude interprets the philosophy algorithmically - provide creative implementation room
- **PURE GENERATIVE ART**: This is about making LIVING ALGORITHMS, not static images with randomness
- **EXPERT CRAFTSMANSHIP**: Repeatedly emphasize the final algorithm must feel meticulously crafted, refined through countless iterations, the product of deep expertise by someone at the absolute top of their field in computational aesthetics

The algorithmic philosophy should be 4-6 paragraphs long. Fill it with poetic computational philosophy that brings together the intended vision. Avoid repeating the same points. Output this algorithmic philosophy as a .md file.

DEDUCING THE CONCEPTUAL SEED

CRITICAL STEP: Before implementing the algorithm, identify the subtle conceptual thread from the original request.

THE ESSENTIAL PRINCIPLE: The concept is a **subtle, niche reference embedded within the algorithm itself** - not always literal, always sophisticated. Someone familiar with the subject should feel it intuitively, while others simply experience a masterful generative composition. The algorithmic philosophy provides the computational language. The deduced concept provides the soul - the quiet conceptual DNA woven invisibly into parameters, behaviors, and emergence patterns.

This is **VERY IMPORTANT**: The reference must be so refined that it enhances the work's depth without announcing itself. Think like a jazz musician quoting another song through algorithmic harmony - only those who know will catch it, but everyone appreciates the generative beauty.

P5.JS IMPLEMENTATION

With the philosophy AND conceptual framework established, express it through code. Pause to gather thoughts before proceeding. Use only the algorithmic philosophy created and the instructions below.

?? STEP 0: READ THE TEMPLATE FIRST ??

CRITICAL: BEFORE writing any HTML:

1. **Read** `templates/viewer.html` using the Read tool
2. **Study** the exact structure, styling, and Anthropic branding
3. **Use that file as the LITERAL STARTING POINT** - not just inspiration
4. **Keep all FIXED sections exactly as shown** (header, sidebar structure, Anthropic colors/fonts, seed controls, action buttons)
5. **Replace only the VARIABLE sections** marked in the file's comments (algorithm, parameters, UI controls for parameters)

Avoid:

- □ Creating HTML from scratch
- □ Inventing custom styling or color schemes
- □ Using system fonts or dark themes
- □ Changing the sidebar structure

Follow these practices:

- □ Copy the template's exact HTML structure
- □ Keep Anthropic branding (Poppins/Lora fonts, light colors, gradient backdrop)
- □ Maintain the sidebar layout (Seed → Parameters → Colors? → Actions)
- □ Replace only the p5.js algorithm and parameter controls

The template is the foundation. Build on it, don't rebuild it.

To create gallery-quality computational art that lives and breathes, use the algorithmic philosophy as the foundation.

TECHNICAL REQUIREMENTS

Seeded Randomness (Art Blocks Pattern):

```
// ALWAYS use a seed for reproducibility
let seed = 12345; // or hash from user input
randomSeed(seed);
noiseSeed(seed);
```

Parameter Structure - FOLLOW THE PHILOSOPHY:

To establish parameters that emerge naturally from the algorithmic philosophy, consider: "What qualities of this system can be adjusted?"

```
let params = {
  seed: 12345,  // Always include seed for reproducibility
  // colors
  // Add parameters that control YOUR algorithm:
  // - Quantities (how many?)
  // - Scales (how big? how fast?)
  // - Probabilities (how likely?)
  // - Ratios (what proportions?)
  // - Angles (what direction?)
  // - Thresholds (when does behavior change?)
};
```

To design effective parameters, focus on the properties the system needs to be tunable rather than thinking in terms of "pattern types".

Core Algorithm - EXPRESS THE PHILOSOPHY:

CRITICAL: The algorithmic philosophy should dictate what to build.

To express the philosophy through code, avoid thinking "which pattern should I use?" and instead think "how to express this philosophy through code?"

If the philosophy is about **organic emergence**, consider using:

- Elements that accumulate or grow over time
- Random processes constrained by natural rules
- Feedback loops and interactions

If the philosophy is about **mathematical beauty**, consider using:

- Geometric relationships and ratios
- Trigonometric functions and harmonics
- Precise calculations creating unexpected patterns

If the philosophy is about **controlled chaos**, consider using:

- Random variation within strict boundaries
- Bifurcation and phase transitions
- Order emerging from disorder

The algorithm flows from the philosophy, not from a menu of options.

To guide the implementation, let the conceptual essence inform creative and original choices. Build something that expresses the vision for this particular request.

Canvas Setup: Standard p5.js structure:

```
function setup() {
  createCanvas(1200, 1200);
  // Initialize your system
}

function draw() {
  // Your generative algorithm
  // Can be static (noLoop) or animated
}
```

CRAFTSMANSHIP REQUIREMENTS

CRITICAL: To achieve mastery, create algorithms that feel like they emerged through countless iterations by a master generative artist. Tune every parameter carefully. Ensure every pattern emerges with purpose. This is NOT random noise - this is CONTROLLED CHAOS refined through deep expertise.

- **Balance**: Complexity without visual noise, order without rigidity
- **Color Harmony**: Thoughtful palettes, not random RGB values
- **Composition**: Even in randomness, maintain visual hierarchy and flow
- **Performance**: Smooth execution, optimized for real-time if animated

- **Reproducibility**: Same seed ALWAYS produces identical output

OUTPUT FORMAT

Output:

1. **Algorithmic Philosophy** - As markdown or text explaining the generative aesthetic
2. **Single HTML Artifact** - Self-contained interactive generative art built from `templates/viewer.html` (see STEP 0 and next section)

The HTML artifact contains everything: p5.js (from CDN), the algorithm, parameter controls, and UI - all in one file that works immediately in claude.ai artifacts or any browser. Start from the template file, not from scratch.

INTERACTIVE ARTIFACT CREATION

REMINDER: `templates/viewer.html` should have already been read (see STEP 0). Use that file as the starting point.

To allow exploration of the generative art, create a single, self-contained HTML artifact. Ensure this artifact works immediately in claude.ai or any browser - no setup required. Embed everything inline.

CRITICAL: WHAT'S FIXED VS VARIABLE

The `templates/viewer.html` file is the foundation. It contains the exact structure and styling needed.

FIXED (always include exactly as shown):

- Layout structure (header, sidebar, main canvas area)
- Anthropic branding (UI colors, fonts, gradients)
- Seed section in sidebar:
  - Seed display
  - Previous/Next buttons
  - Random button
  - Jump to seed input + Go button
- Actions section in sidebar:
  - Regenerate button
  - Reset button

VARIABLE (customize for each artwork):

- The entire p5.js algorithm (setup/draw/classes)

- The parameters object (define what the art needs)
- The Parameters section in sidebar:
  - Number of parameter controls
  - Parameter names
  - Min/max/step values for sliders
  - Control types (sliders, inputs, etc.)
- Colors section (optional):
  - Some art needs color pickers
  - Some art might use fixed colors
  - Some art might be monochrome (no color controls needed)
  - Decide based on the art's needs

Every artwork should have unique parameters and algorithm! The fixed parts provide consistent UX - everything else expresses the unique vision.

REQUIRED FEATURES

1. Parameter Controls

- Sliders for numeric parameters (particle count, noise scale, speed, etc.)
- Color pickers for palette colors
- Real-time updates when parameters change
- Reset button to restore defaults

2. Seed Navigation

- Display current seed number
- "Previous" and "Next" buttons to cycle through seeds
- "Random" button for random seed
- Input field to jump to specific seed
- Generate 100 variations when requested (seeds 1-100)

3. Single Artifact Structure

```
<!DOCTYPE html>
<html>
<head>
  <!-- p5.js from CDN - always available -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/1.7.0/p5.min.js"></script>
  <style>
    /* All styling inline - clean, minimal */
    /* Canvas on top, controls below */
  </style>
</head>
```

```
<body>
  <div id="canvas-container"></div>
  <div id="controls">
    <!-- All parameter controls -->
  </div>
  <script>
    // ALL p5.js code inline here
    // Parameter objects, classes, functions
    // setup() and draw()
    // UI handlers
    // Everything self-contained
  </script>
</body>
</html>
```

CRITICAL: This is a single artifact. No external files, no imports (except p5.js CDN). Everything inline.

4. Implementation Details - BUILD THE SIDEBAR

The sidebar structure:

1. Seed (FIXED) - Always include exactly as shown:

- Seed display
- Prev/Next/Random/Jump buttons

2. Parameters (VARIABLE) - Create controls for the art:

```
<div class="control-group">
    <label>Parameter Name</label>
    <input type="range" id="param" min="..." max="..." step="..." value="..."
oninput="updateParam('param', this.value)">
    <span class="value-display" id="param-value">...</span>
</div>
```

Add as many control-group divs as there are parameters.

3. Colors (OPTIONAL/VARIABLE) - Include if the art needs adjustable colors:

- Add color pickers if users should control palette
- Skip this section if the art uses fixed colors
- Skip if the art is monochrome

4. Actions (FIXED) - Always include exactly as shown:

- Regenerate button
- Reset button
- Download PNG button

Requirements:

- Seed controls must work (prev/next/random/jump/display)
- All parameters must have UI controls
- Regenerate, Reset, Download buttons must work
- Keep Anthropic branding (UI styling, not art colors)

USING THE ARTIFACT

The HTML artifact works immediately:

1. **In claude.ai**: Displayed as an interactive artifact - runs instantly
2. **As a file**: Save and open in any browser - no server needed
3. **Sharing**: Send the HTML file - it's completely self-contained

---

VARIATIONS & EXPLORATION

The artifact includes seed navigation by default (prev/next/random buttons), allowing users to explore variations without creating multiple files. If the user wants specific variations highlighted:

- Include seed presets (buttons for "Variation 1: Seed 42", "Variation 2: Seed 127", etc.)
- Add a "Gallery Mode" that shows thumbnails of multiple seeds side-by-side
- All within the same single artifact

This is like creating a series of prints from the same plate - the algorithm is consistent, but each seed reveals different facets of its potential. The interactive nature means users discover their own favorites by exploring the seed space.

THE CREATIVE PROCESS

User request → Algorithmic philosophy → Implementation

Each request is unique. The process involves:

1. **Interpret the user's intent** - What aesthetic is being sought?

2. **Create an algorithmic philosophy** (4-6 paragraphs) describing the computational approach
3. **Implement it in code** - Build the algorithm that expresses this philosophy
4. **Design appropriate parameters** - What should be tunable?
5. **Build matching UI controls** - Sliders/inputs for those parameters

The constants:

- Anthropic branding (colors, fonts, layout)
- Seed navigation (always present)
- Self-contained HTML artifact

Everything else is variable:

- The algorithm itself
- The parameters
- The UI controls
- The visual outcome

To achieve the best results, trust creativity and let the philosophy guide the implementation.

RESOURCES

This skill includes helpful templates and documentation:

- **templates/viewer.html**: REQUIRED STARTING POINT for all HTML artifacts.
  - This is the foundation - contains the exact structure and Anthropic branding
  - **Keep unchanged**: Layout structure, sidebar organization, Anthropic colors/fonts, seed controls, action buttons
  - **Replace**: The p5.js algorithm, parameter definitions, and UI controls in Parameters section
  - The extensive comments in the file mark exactly what to keep vs replace
- **templates/generator_template.js**: Reference for p5.js best practices and code structure principles.
  - Shows how to organize parameters, use seeded randomness, structure classes
  - NOT a pattern menu - use these principles to build unique algorithms
  - Embed algorithms inline in the HTML artifact (don't create separate .js files)

Critical reminder:

- The **template is the STARTING POINT**, not inspiration
- The **algorithm is where to create** something unique
- Don't copy the flow field example - build what the philosophy demands
- But DO keep the exact UI structure and Anthropic branding from the template

/internal-comms

Source: `~/.claude/skills/internal-comms/SKILL.md`

---

name: internal-comms description: A set of resources to help me write all kinds of internal communications, using the formats that my company likes to use. Claude should use this skill whenever asked to write some sort of internal communications (status reports, leadership updates, 3P updates, company newsletters, FAQs, incident reports, project updates, etc.). license: Complete terms in LICENSE.txt

When to use this skill

To write internal communications, use this skill for:

- 3P updates (Progress, Plans, Problems)

- Company newsletters
- FAQ responses
- Status reports
- Leadership updates
- Project updates
- Incident reports

How to use this skill

To write any internal communication:

1. **Identify the communication type** from the request
2. **Load the appropriate guideline file** from the `examples/` directory:
   - `examples/3p-updates.md` - For Progress/Plans/Problems team updates
   - `examples/company-newsletter.md` - For company-wide newsletters
   - `examples/faq-answers.md` - For answering frequently asked questions
   - `examples/general-comms.md` - For anything else that doesn't explicitly match one of the above
3. **Follow the specific instructions** in that file for formatting, tone, and content gathering

If the communication type doesn't match any existing guideline, ask for clarification or more context about the desired format.

Keywords

3P updates, company newsletter, company comms, weekly update, faqs, common questions, updates, internal comms

/mcp-builder

Source: `~/.claude/skills/mcp-builder/SKILL.md`

---

name: mcp-builder description: Guide for creating high-quality MCP (Model Context Protocol) servers that enable LLMs to interact with external services through well-designed tools. Use when building MCP servers to integrate external APIs or services, whether in Python (FastMCP) or Node/TypeScript (MCP SDK). license: Complete terms in LICENSE.txt

MCP Server Development Guide

Overview

Create MCP (Model Context Protocol) servers that enable LLMs to interact with external services through well-designed tools. The quality of an MCP server is measured by how well it enables LLMs

to accomplish real-world tasks.

Process

? High-Level Workflow

Creating a high-quality MCP server involves four main phases:

Phase 1: Deep Research and Planning

1.1 Understand Modern MCP Design

API Coverage vs. Workflow Tools: Balance comprehensive API endpoint coverage with specialized workflow tools. Workflow tools can be more convenient for specific tasks, while comprehensive coverage gives agents flexibility to compose operations. Performance varies by client—some clients benefit from code execution that combines basic tools, while others work better with higher-level workflows. When uncertain, prioritize comprehensive API coverage.

Tool Naming and Discoverability: Clear, descriptive tool names help agents find the right tools quickly. Use consistent prefixes (e.g., `github_create_issue`, `github_list_repos`) and action-oriented naming.

Context Management: Agents benefit from concise tool descriptions and the ability to filter/paginate results. Design tools that return focused, relevant data. Some clients support code execution which can help agents filter and process data efficiently.

Actionable Error Messages: Error messages should guide agents toward solutions with specific suggestions and next steps.

1.2 Study MCP Protocol Documentation

Navigate the MCP specification:

Start with the sitemap to find relevant pages: `https://modelcontextprotocol.io/sitemap.xml`

Then fetch specific pages with `.md` suffix for markdown format (e.g., `https://modelcontextprotocol.io/specification/draft.md`).

Key pages to review:

- Specification overview and architecture
- Transport mechanisms (streamable HTTP, stdio)

- Tool, resource, and prompt definitions

1.3 Study Framework Documentation

Recommended stack:

- **Language**: TypeScript (high-quality SDK support and good compatibility in many execution environments e.g. MCPB. Plus AI models are good at generating TypeScript code, benefiting from its broad usage, static typing and good linting tools)
- **Transport**: Streamable HTTP for remote servers, using stateless JSON (simpler to scale and maintain, as opposed to stateful sessions and streaming responses). stdio for local servers.

Load framework documentation:

- **MCP Best Practices**: □□ View Best Practices - Core guidelines

For TypeScript (recommended):

- **TypeScript SDK**: Use WebFetch to load `https://raw.githubusercontent.com/modelcontextprotocol/typescript-sdk/main/README.md`
- ⚡ TypeScript Guide - TypeScript patterns and examples

For Python:

- **Python SDK**: Use WebFetch to load `https://raw.githubusercontent.com/modelcontextprotocol/python-sdk/main/README.md`
- □□ Python Guide - Python patterns and examples

1.4 Plan Your Implementation

Understand the API: Review the service's API documentation to identify key endpoints, authentication requirements, and data models. Use web search and WebFetch as needed.

Tool Selection: Prioritize comprehensive API coverage. List endpoints to implement, starting with the most common operations.

Phase 2: Implementation

2.1 Set Up Project Structure

See language-specific guides for project setup:

- ⚡ TypeScript Guide - Project structure, package.json, tsconfig.json

- 📖 Python Guide - Module organization, dependencies

2.2 Implement Core Infrastructure

Create shared utilities:

- API client with authentication
- Error handling helpers
- Response formatting (JSON/Markdown)
- Pagination support

2.3 Implement Tools

For each tool:

Input Schema:

- Use Zod (TypeScript) or Pydantic (Python)
- Include constraints and clear descriptions
- Add examples in field descriptions

Output Schema:

- Define `outputSchema` where possible for structured data
- Use `structuredContent` in tool responses (TypeScript SDK feature)
- Helps clients understand and process tool outputs

Tool Description:

- Concise summary of functionality
- Parameter descriptions
- Return type schema

Implementation:

- Async/await for I/O operations
- Proper error handling with actionable messages
- Support pagination where applicable
- Return both text content and structured data when using modern SDKs

Annotations:

- `readOnlyHint`: true/false
- `destructiveHint`: true/false
- `idempotentHint`: true/false
- `openWorldHint`: true/false

Phase 3: Review and Test

3.1 Code Quality

Review for:

- No duplicated code (DRY principle)
- Consistent error handling
- Full type coverage
- Clear tool descriptions

3.2 Build and Test

TypeScript:

- Run `npm run build` to verify compilation
- Test with MCP Inspector: `npx @modelcontextprotocol/inspector`

Python:

- Verify syntax: `python -m py_compile your_server.py`
- Test with MCP Inspector

See language-specific guides for detailed testing approaches and quality checklists.

Phase 4: Create Evaluations

After implementing your MCP server, create comprehensive evaluations to test its effectiveness.

Load Evaluation Guide for complete evaluation guidelines.

4.1 Understand Evaluation Purpose

Use evaluations to test whether LLMs can effectively use your MCP server to answer realistic, complex questions.

4.2 Create 10 Evaluation Questions

To create effective evaluations, follow the process outlined in the evaluation guide:

1. **Tool Inspection**: List available tools and understand their capabilities
2. **Content Exploration**: Use READ-ONLY operations to explore available data
3. **Question Generation**: Create 10 complex, realistic questions
4. **Answer Verification**: Solve each question yourself to verify answers

4.3 Evaluation Requirements

Ensure each question is:

- **Independent**: Not dependent on other questions
- **Read-only**: Only non-destructive operations required
- **Complex**: Requiring multiple tool calls and deep exploration
- **Realistic**: Based on real use cases humans would care about
- **Verifiable**: Single, clear answer that can be verified by string comparison
- **Stable**: Answer won't change over time

4.4 Output Format

Create an XML file with this structure:

```
<evaluation>
  <qa_pair>
    <question>Find discussions about AI model launches with animal codenames. One model needed a specific safety designation that uses the format ASL-X. What number X was being determined for the model named after a spotted wild cat?</question>
    <answer>3</answer>
  </qa_pair>
<!-- More qa_pairs... -->
</evaluation>
```

Reference Files

? Documentation Library

Load these resources as needed during development:

Core MCP Documentation (Load First)

- **MCP Protocol**: Start with sitemap at `https://modelcontextprotocol.io/sitemap.xml`, then fetch specific pages with `.md` suffix
- MCP Best Practices - Universal MCP guidelines including:
  - Server and tool naming conventions
  - Response format guidelines (JSON vs Markdown)
  - Pagination best practices

- Transport selection (streamable HTTP vs stdio)
- Security and error handling standards

SDK Documentation (Load During Phase 1/2)

- **Python SDK**: Fetch from `https://raw.githubusercontent.com/modelcontextprotocol/python-sdk/main/README.md`
- **TypeScript SDK**: Fetch from `https://raw.githubusercontent.com/modelcontextprotocol/typescript-sdk/main/README.md`

Language-Specific Implementation Guides (Load During Phase 2)

- Python Implementation Guide - Complete Python/FastMCP guide with:
  - Server initialization patterns
  - Pydantic model examples
  - Tool registration with `@mcp.tool`
  - Complete working examples
  - Quality checklist
- ⚡ TypeScript Implementation Guide - Complete TypeScript guide with:
  - Project structure
  - Zod schema patterns
  - Tool registration with `server.registerTool`
  - Complete working examples
  - Quality checklist

Evaluation Guide (Load During Phase 4)

- Evaluation Guide - Complete evaluation creation guide with:
  - Question creation guidelines
  - Answer verification strategies
  - XML format specifications
  - Example questions and answers
  - Running an evaluation with the provided scripts

/skill-creator

Source: `~/.claude/skills/skill-creator/SKILL.md`

---

name: skill-creator version: "2.0" level: 3 trigger: "create skill, new skill, update skill, skill creator, SKILL.md" author: john updated: 2026-03-16 description: Guide for creating Level 3+ skills. IF new skill request THEN scaffold SKILL.md with metadata + if/then workflow + verification + MAX TURNS. Update skill-registry.db on completion. license: Complete terms in LICENSE.txt

Skill Creator

This skill provides guidance for creating effective skills.

About Skills

Skills are modular, self-contained packages that extend Claude's capabilities by providing specialized knowledge, workflows, and tools. Think of them as "onboarding guides" for specific domains or tasks—they transform Claude from a general-purpose agent into a specialized agent equipped with procedural knowledge that no model can fully possess.

What Skills Provide

1. Specialized workflows - Multi-step procedures for specific domains
2. Tool integrations - Instructions for working with specific file formats or APIs
3. Domain expertise - Company-specific knowledge, schemas, business logic
4. Bundled resources - Scripts, references, and assets for complex and repetitive tasks

Core Principles

Concise is Key

The context window is a public good. Skills share the context window with everything else Claude needs: system prompt, conversation history, other Skills' metadata, and the actual user request.

Default assumption: Claude is already very smart. Only add context Claude doesn't already have. Challenge each piece of information: "Does Claude really need this explanation?" and "Does this paragraph justify its token cost?"

Prefer concise examples over verbose explanations.

Set Appropriate Degrees of Freedom

Match the level of specificity to the task's fragility and variability:

High freedom (text-based instructions): Use when multiple approaches are valid, decisions depend on context, or heuristics guide the approach.

Medium freedom (pseudocode or scripts with parameters): Use when a preferred pattern exists, some variation is acceptable, or configuration affects behavior.

Low freedom (specific scripts, few parameters): Use when operations are fragile and error-prone, consistency is critical, or a specific sequence must be followed.

Think of Claude as exploring a path: a narrow bridge with cliffs needs specific guardrails (low freedom), while an open field allows many routes (high freedom).

Anatomy of a Skill

Every skill consists of a required SKILL.md file and optional bundled resources:

```
skill-name/
├── SKILL.md (required)
│   ├── YAML frontmatter metadata (required)
│   │   ├── name: (required)
│   │   ├── description: (required)
│   │   └── compatibility: (optional, rarely needed)
│   └── Markdown instructions (required)
└── Bundled Resources (optional)
    ├── scripts/          - Executable code (Python/Bash/etc.)
    ├── references/       - Documentation intended to be loaded into context as needed
    └── assets/           - Files used in output (templates, icons, fonts, etc.)
```

SKILL.md (required)

Every SKILL.md consists of:

- **Frontmatter** (YAML): Contains `name` and `description` fields (required), plus optional fields like `license`, `metadata`, and `compatibility`. Only `name` and `description` are read by Claude to determine when the skill triggers, so be clear and comprehensive about what the skill is and when it should be used. The `compatibility` field is for noting environment requirements (target product, system packages, etc.) but most skills don't need it.
- **Body** (Markdown): Instructions and guidance for using the skill. Only loaded AFTER the skill triggers (if at all).

Bundled Resources (optional)

Scripts (`scripts/`)

Executable code (Python/Bash/etc.) for tasks that require deterministic reliability or are repeatedly rewritten.

- **When to include**: When the same code is being rewritten repeatedly or deterministic reliability is needed
- **Example**: `scripts/rotate_pdf.py` for PDF rotation tasks
- **Benefits**: Token efficient, deterministic, may be executed without loading into context
- **Note**: Scripts may still need to be read by Claude for patching or environment-specific adjustments

References (`references/`)

Documentation and reference material intended to be loaded as needed into context to inform Claude's process and thinking.

- **When to include**: For documentation that Claude should reference while working
- **Examples**: `references/finance.md` for financial schemas, `references/mnda.md` for company NDA template, `references/policies.md` for company policies, `references/api_docs.md` for API specifications
- **Use cases**: Database schemas, API documentation, domain knowledge, company policies, detailed workflow guides
- **Benefits**: Keeps SKILL.md lean, loaded only when Claude determines it's needed
- **Best practice**: If files are large (>10k words), include grep search patterns in SKILL.md
- **Avoid duplication**: Information should live in either SKILL.md or references files, not both. Prefer references files for detailed information unless it's truly core to the skill—this keeps SKILL.md lean while making information discoverable without hogging the context window. Keep only essential procedural instructions and workflow guidance in SKILL.md; move detailed reference material, schemas, and examples to references files.

Assets (`assets/`)

Files not intended to be loaded into context, but rather used within the output Claude produces.

- **When to include**: When the skill needs files that will be used in the final output
- **Examples**: `assets/logo.png` for brand assets, `assets/slides.pptx` for PowerPoint templates, `assets/frontend-template/` for HTML/React boilerplate, `assets/font.ttf` for typography
- **Use cases**: Templates, images, icons, boilerplate code, fonts, sample documents that get copied or modified
- **Benefits**: Separates output resources from documentation, enables Claude to use files without loading them into context

What to Not Include in a Skill

A skill should only contain essential files that directly support its functionality. Do NOT create extraneous documentation or auxiliary files, including:

- README.md
- INSTALLATION_GUIDE.md
- QUICK_REFERENCE.md
- CHANGELOG.md
- etc.

The skill should only contain the information needed for an AI agent to do the job at hand. It should not contain auxilary context about the process that went into creating it, setup and testing procedures, user-facing documentation, etc. Creating additional documentation files just adds clutter and confusion.

Progressive Disclosure Design Principle

Skills use a three-level loading system to manage context efficiently:

1. **Metadata (name + description)** - Always in context (~100 words)
2. **SKILL.md body** - When skill triggers (<5k words)
3. **Bundled resources** - As needed by Claude (Unlimited because scripts can be executed without reading into context window)

Progressive Disclosure Patterns

Keep SKILL.md body to the essentials and under 500 lines to minimize context bloat. Split content into separate files when approaching this limit. When splitting out content into other files, it is very important to reference them from SKILL.md and describe clearly when to read them, to ensure the reader of the skill knows they exist and when to use them.

Key principle: When a skill supports multiple variations, frameworks, or options, keep only the core workflow and selection guidance in SKILL.md. Move variant-specific details (patterns, examples, configuration) into separate reference files.

Pattern 1: High-level guide with references

```
# PDF Processing

## Quick start

Extract text with pdfplumber:
[code example]

## Advanced features

- **Form filling**: See [FORMS.md](FORMS.md) for complete guide
- **API reference**: See [REFERENCE.md](REFERENCE.md) for all methods
- **Examples**: See [EXAMPLES.md](EXAMPLES.md) for common patterns
```

Claude loads FORMS.md, REFERENCE.md, or EXAMPLES.md only when needed.

Pattern 2: Domain-specific organization

For Skills with multiple domains, organize content by domain to avoid loading irrelevant context:

```
bigquery-skill/
├── SKILL.md (overview and navigation)
└── reference/
    ├── finance.md (revenue, billing metrics)
    ├── sales.md (opportunities, pipeline)
    ├── product.md (API usage, features)
```

```
    └── marketing.md (campaigns, attribution)
```

When a user asks about sales metrics, Claude only reads sales.md.

Similarly, for skills supporting multiple frameworks or variants, organize by variant:

```
cloud-deploy/
├── SKILL.md (workflow + provider selection)
└── references/
    ├── aws.md (AWS deployment patterns)
    ├── gcp.md (GCP deployment patterns)
    └── azure.md (Azure deployment patterns)
```

When the user chooses AWS, Claude only reads aws.md.

Pattern 3: Conditional details

Show basic content, link to advanced content:

```
# DOCX Processing

## Creating documents

Use docx-js for new documents. See [DOCX-JS.md](DOCX-JS.md).

## Editing documents

For simple edits, modify the XML directly.

**For tracked changes**: See [REDLINING.md](REDLINING.md)
**For OOXML details**: See [OOXML.md](OOXML.md)
```

Claude reads REDLINING.md or OOXML.md only when the user needs those features.

Important guidelines:

- **Avoid deeply nested references** - Keep references one level deep from SKILL.md. All reference files should link directly from SKILL.md.
- **Structure longer reference files** - For files longer than 100 lines, include a table of contents at the top so Claude can see the full scope when previewing.

Skill Creation Process

Skill creation involves these steps:

1. Understand the skill with concrete examples
2. Plan reusable skill contents (scripts, references, assets)
3. Initialize the skill (run init_skill.py)
4. Edit the skill (implement resources and write SKILL.md)
5. Package the skill (run package_skill.py)
6. Iterate based on real usage

Follow these steps in order, skipping only if there is a clear reason why they are not applicable.

Step 1: Understanding the Skill with Concrete Examples

Skip this step only when the skill's usage patterns are already clearly understood. It remains valuable even when working with an existing skill.

To create an effective skill, clearly understand concrete examples of how the skill will be used. This understanding can come from either direct user examples or generated examples that are validated with user feedback.

For example, when building an image-editor skill, relevant questions include:

- "What functionality should the image-editor skill support? Editing, rotating, anything else?"
- "Can you give some examples of how this skill would be used?"
- "I can imagine users asking for things like 'Remove the red-eye from this image' or 'Rotate this image'. Are there other ways you imagine this skill being used?"
- "What would a user say that should trigger this skill?"

To avoid overwhelming users, avoid asking too many questions in a single message. Start with the most important questions and follow up as needed for better effectiveness.

Conclude this step when there is a clear sense of the functionality the skill should support.

Step 2: Planning the Reusable Skill Contents

To turn concrete examples into an effective skill, analyze each example by:

1. Considering how to execute on the example from scratch
2. Identifying what scripts, references, and assets would be helpful when executing these workflows repeatedly

Example: When building a `pdf-editor` skill to handle queries like "Help me rotate this PDF," the analysis shows:

1. Rotating a PDF requires re-writing the same code each time
2. A `scripts/rotate_pdf.py` script would be helpful to store in the skill

Example: When designing a `frontend-webapp-builder` skill for queries like "Build me a todo app" or "Build me a dashboard to track my steps," the analysis shows:

1. Writing a frontend webapp requires the same boilerplate HTML/React each time
2. An `assets/hello-world/` template containing the boilerplate HTML/React project files would be helpful to store in the skill

Example: When building a `big-query` skill to handle queries like "How many users have logged in today?" the analysis shows:

1. Querying BigQuery requires re-discovering the table schemas and relationships each time
2. A `references/schema.md` file documenting the table schemas would be helpful to store in the skill

To establish the skill's contents, analyze each concrete example to create a list of the reusable resources to include: scripts, references, and assets.

Step 3: Initializing the Skill

At this point, it is time to actually create the skill.

Skip this step only if the skill being developed already exists, and iteration or packaging is needed. In this case, continue to the next step.

When creating a new skill from scratch, always run the `init_skill.py` script. The script conveniently generates a new template skill directory that automatically includes everything a skill requires, making the skill creation process much more efficient and reliable.

Usage:

```
scripts/init_skill.py <skill-name> --path <output-directory>
```

The script:

- Creates the skill directory at the specified path
- Generates a SKILL.md template with proper frontmatter and TODO placeholders
- Creates example resource directories: `scripts/`, `references/`, and `assets/`
- Adds example files in each directory that can be customized or deleted

After initialization, customize or remove the generated SKILL.md and example files as needed.

Step 4: Edit the Skill

When editing the (newly-generated or existing) skill, remember that the skill is being created for another instance of Claude to use. Include information that would be beneficial and non-obvious to Claude. Consider what procedural knowledge, domain-specific details, or reusable assets would help another Claude instance execute these tasks more effectively.

Learn Proven Design Patterns

Consult these helpful guides based on your skill's needs:

- **Multi-step processes**: See references/workflows.md for sequential workflows and conditional logic
- **Specific output formats or quality standards**: See references/output-patterns.md for template and example patterns

These files contain established best practices for effective skill design.

Start with Reusable Skill Contents

To begin implementation, start with the reusable resources identified above: `scripts/`, `references/`, and `assets/` files. Note that this step may require user input. For example, when implementing a `brand-guidelines` skill, the user may need to provide brand assets or templates to store in `assets/`, or documentation to store in `references/`.

Added scripts must be tested by actually running them to ensure there are no bugs and that the output matches what is expected. If there are many similar scripts, only a representative sample needs to be tested to ensure confidence that they all work while balancing time to completion.

Any example files and directories not needed for the skill should be deleted. The initialization script creates example files in `scripts/`, `references/`, and `assets/` to demonstrate structure, but most skills won't need all of them.

Update SKILL.md

Writing Guidelines: Always use imperative/infinitive form.

Frontmatter

Write the YAML frontmatter with `name` and `description`:

- `name`: The skill name
- `description`: This is the primary triggering mechanism for your skill, and helps Claude understand when to use the skill.
  - Include both what the Skill does and specific triggers/contexts for when to use it.
  - Include all "when to use" information here - Not in the body. The body is only loaded after triggering, so "When to Use This Skill" sections in the body are not helpful to

Claude.
    - Example description for a `docx` skill: "Comprehensive document creation, editing, and analysis with support for tracked changes, comments, formatting preservation, and text extraction. Use when Claude needs to work with professional documents (.docx files) for: (1) Creating new documents, (2) Modifying or editing content, (3) Working with tracked changes, (4) Adding comments, or any other document tasks"

Do not include any other fields in YAML frontmatter.

Body

Write instructions for using the skill and its bundled resources.

Step 5: Packaging a Skill

Once development of the skill is complete, it must be packaged into a distributable .skill file that gets shared with the user. The packaging process automatically validates the skill first to ensure it meets all requirements:

```
scripts/package_skill.py <path/to/skill-folder>
```

Optional output directory specification:

```
scripts/package_skill.py <path/to/skill-folder> ./dist
```

The packaging script will:

1. **Validate** the skill automatically, checking:
    - YAML frontmatter format and required fields
    - Skill naming conventions and directory structure
    - Description completeness and quality
    - File organization and resource references
2. **Package** the skill if validation passes, creating a .skill file named after the skill (e.g., `my-skill.skill`) that includes all files and maintains the proper directory structure for distribution. The .skill file is a zip file with a .skill extension.

If validation fails, the script will report the errors and exit without creating a package. Fix any validation errors and run the packaging command again.

Step 6: Iterate

After testing the skill, users may request improvements. Often this happens right after using the skill, with fresh context of how the skill performed.

Iteration workflow:

1. Use the skill on real tasks
2. Notice struggles or inefficiencies
3. Identify how SKILL.md or bundled resources should be updated
4. Implement changes and test again

/slack-gif-creator

Source: `~/.claude/skills/slack-gif-creator/SKILL.md`

---

name: slack-gif-creator description: Knowledge and utilities for creating animated GIFs optimized for Slack. Provides constraints, validation tools, and animation concepts. Use when users request animated GIFs for Slack like "make me a GIF of X doing Y for Slack." license: Complete terms in LICENSE.txt

Slack GIF Creator

A toolkit providing utilities and knowledge for creating animated GIFs optimized for Slack.

Slack Requirements

Dimensions:

- Emoji GIFs: 128x128 (recommended)

- Message GIFs: 480x480

Parameters:

- FPS: 10-30 (lower is smaller file size)
- Colors: 48-128 (fewer = smaller file size)
- Duration: Keep under 3 seconds for emoji GIFs

Core Workflow

```
from core.gif_builder import GIFBuilder
from PIL import Image, ImageDraw

# 1. Create builder
builder = GIFBuilder(width=128, height=128, fps=10)

# 2. Generate frames
for i in range(12):
    frame = Image.new('RGB', (128, 128), (240, 248, 255))
    draw = ImageDraw.Draw(frame)

    # Draw your animation using PIL primitives
    # (circles, polygons, lines, etc.)

    builder.add_frame(frame)

# 3. Save with optimization
builder.save('output.gif', num_colors=48, optimize_for_emoji=True)
```

Drawing Graphics

Working with User-Uploaded Images

If a user uploads an image, consider whether they want to:

- **Use it directly** (e.g., "animate this", "split this into frames")
- **Use it as inspiration** (e.g., "make something like this")

Load and work with images using PIL:

```
from PIL import Image

uploaded = Image.open('file.png')
# Use directly, or just as reference for colors/style
```

Drawing from Scratch

When drawing graphics from scratch, use PIL ImageDraw primitives:

```
from PIL import ImageDraw

draw = ImageDraw.Draw(frame)

# Circles/ovals
draw.ellipse([x1, y1, x2, y2], fill=(r, g, b), outline=(r, g, b), width=3)

# Stars, triangles, any polygon
points = [(x1, y1), (x2, y2), (x3, y3), ...]
draw.polygon(points, fill=(r, g, b), outline=(r, g, b), width=3)

# Lines
draw.line([(x1, y1), (x2, y2)], fill=(r, g, b), width=5)

# Rectangles
draw.rectangle([x1, y1, x2, y2], fill=(r, g, b), outline=(r, g, b), width=3)
```

Don't use: Emoji fonts (unreliable across platforms) or assume pre-packaged graphics exist in this skill.

Making Graphics Look Good

Graphics should look polished and creative, not basic. Here's how:

Use thicker lines - Always set `width=2` or higher for outlines and lines. Thin lines (width=1) look choppy and amateurish.

Add visual depth:

- Use gradients for backgrounds (`create_gradient_background`)
- Layer multiple shapes for complexity (e.g., a star with a smaller star inside)

Make shapes more interesting:

- Don't just draw a plain circle - add highlights, rings, or patterns
- Stars can have glows (draw larger, semi-transparent versions behind)
- Combine multiple shapes (stars + sparkles, circles + rings)

Pay attention to colors:

- Use vibrant, complementary colors
- Add contrast (dark outlines on light shapes, light outlines on dark shapes)
- Consider the overall composition

For complex shapes (hearts, snowflakes, etc.):

- Use combinations of polygons and ellipses
- Calculate points carefully for symmetry
- Add details (a heart can have a highlight curve, snowflakes have intricate branches)

Be creative and detailed! A good Slack GIF should look polished, not like placeholder graphics.

Available Utilities

GIFBuilder (`core.gif_builder`)

Assembles frames and optimizes for Slack:

```
builder = GIFBuilder(width=128, height=128, fps=10)
builder.add_frame(frame)  # Add PIL Image
builder.add_frames(frames)  # Add list of frames
builder.save('out.gif', num_colors=48, optimize_for_emoji=True, remove_duplicates=True)
```

Validators (`core.validators`)

Check if GIF meets Slack requirements:

```
from core.validators import validate_gif, is_slack_ready

# Detailed validation
passes, info = validate_gif('my.gif', is_emoji=True, verbose=True)

# Quick check
```

```
if is_slack_ready('my.gif'):
    print("Ready!")
```

Easing Functions (`core.easing`)

Smooth motion instead of linear:

```
from core.easing import interpolate

# Progress from 0.0 to 1.0
t = i / (num_frames - 1)

# Apply easing
y = interpolate(start=0, end=400, t=t, easing='ease_out')

# Available: linear, ease_in, ease_out, ease_in_out,
#            bounce_out, elastic_out, back_out
```

Frame Helpers (`core.frame_composer`)

Convenience functions for common needs:

```
from core.frame_composer import (
    create_blank_frame,         # Solid color background
    create_gradient_background,  # Vertical gradient
    draw_circle,                # Helper for circles
    draw_text,                  # Simple text rendering
    draw_star                   # 5-pointed star
)
```

Animation Concepts

Shake/Vibrate

Offset object position with oscillation:

- Use `math.sin()` or `math.cos()` with frame index
- Add small random variations for natural feel

- Apply to x and/or y position

Pulse/Heartbeat

Scale object size rhythmically:

- Use `math.sin(t * frequency * 2 * math.pi)` for smooth pulse
- For heartbeat: two quick pulses then pause (adjust sine wave)
- Scale between 0.8 and 1.2 of base size

Bounce

Object falls and bounces:

- Use `interpolate()` with `easing='bounce_out'` for landing
- Use `easing='ease_in'` for falling (accelerating)
- Apply gravity by increasing y velocity each frame

Spin/Rotate

Rotate object around center:

- PIL: `image.rotate(angle, resample=Image.BICUBIC)`
- For wobble: use sine wave for angle instead of linear

Fade In/Out

Gradually appear or disappear:

- Create RGBA image, adjust alpha channel
- Or use `Image.blend(image1, image2, alpha)`
- Fade in: alpha from 0 to 1
- Fade out: alpha from 1 to 0

Slide

Move object from off-screen to position:

- Start position: outside frame bounds
- End position: target location
- Use `interpolate()` with `easing='ease_out'` for smooth stop
- For overshoot: use `easing='back_out'`

Zoom

Scale and position for zoom effect:

- Zoom in: scale from 0.1 to 2.0, crop center
- Zoom out: scale from 2.0 to 1.0
- Can add motion blur for drama (PIL filter)

Explode/Particle Burst

Create particles radiating outward:

- Generate particles with random angles and velocities
- Update each particle: `x += vx`, `y += vy`
- Add gravity: `vy += gravity_constant`
- Fade out particles over time (reduce alpha)

Optimization Strategies

Only when asked to make the file size smaller, implement a few of the following methods:

1. **Fewer frames** - Lower FPS (10 instead of 20) or shorter duration
2. **Fewer colors** - `num_colors=48` instead of 128
3. **Smaller dimensions** - 128x128 instead of 480x480
4. **Remove duplicates** - `remove_duplicates=True` in save()
5. **Emoji mode** - `optimize_for_emoji=True` auto-optimizes

```
# Maximum optimization for emoji
builder.save(
    'emoji.gif',
    num_colors=48,
    optimize_for_emoji=True,
    remove_duplicates=True
)
```

Philosophy

This skill provides:

- **Knowledge**: Slack's requirements and animation concepts

- **Utilities**: GIFBuilder, validators, easing functions
- **Flexibility**: Create the animation logic using PIL primitives

It does NOT provide:

- Rigid animation templates or pre-made functions
- Emoji font rendering (unreliable across platforms)
- A library of pre-packaged graphics built into the skill

Note on user uploads: This skill doesn't include pre-built graphics, but if a user uploads an image, use PIL to load and work with it - interpret based on their request whether they want it used directly or just as inspiration.

Be creative! Combine concepts (bouncing + rotating, pulsing + sliding, etc.) and use PIL's full capabilities.

Dependencies

```
pip install pillow imageio numpy
```

/theme-factory

Source: `~/.claude/skills/theme-factory/SKILL.md`

---

name: theme-factory description: Toolkit for styling artifacts with a theme. These artifacts can be slides, docs, reportings, HTML landing pages, etc. There are 10 pre-set themes with colors/fonts that you can apply to any artifact that has been creating, or can generate a new theme on-the-fly. license: Complete terms in LICENSE.txt

Theme Factory Skill

This skill provides a curated collection of professional font and color themes themes, each with carefully selected color palettes and font pairings. Once a theme is chosen, it can be applied to any artifact.

Purpose

To apply consistent, professional styling to presentation slide decks, use this skill. Each theme includes:

- A cohesive color palette with hex codes
- Complementary font pairings for headers and body text
- A distinct visual identity suitable for different contexts and audiences

Usage Instructions

To apply styling to a slide deck or other artifact:

1. **Show the theme showcase**: Display the `theme-showcase.pdf` file to allow users to see all available themes visually. Do not make any modifications to it; simply show the file for viewing.
2. **Ask for their choice**: Ask which theme to apply to the deck
3. **Wait for selection**: Get explicit confirmation about the chosen theme
4. **Apply the theme**: Once a theme has been chosen, apply the selected theme's colors and fonts to the deck/artifact

Themes Available

The following 10 themes are available, each showcased in `theme-showcase.pdf`:

1. **Ocean Depths** - Professional and calming maritime theme
2. **Sunset Boulevard** - Warm and vibrant sunset colors
3. **Forest Canopy** - Natural and grounded earth tones
4. **Modern Minimalist** - Clean and contemporary grayscale
5. **Golden Hour** - Rich and warm autumnal palette
6. **Arctic Frost** - Cool and crisp winter-inspired theme
7. **Desert Rose** - Soft and sophisticated dusty tones
8. **Tech Innovation** - Bold and modern tech aesthetic
9. **Botanical Garden** - Fresh and organic garden colors
10. **Midnight Galaxy** - Dramatic and cosmic deep tones

Theme Details

Each theme is defined in the `themes/` directory with complete specifications including:

- Cohesive color palette with hex codes
- Complementary font pairings for headers and body text
- Distinct visual identity suitable for different contexts and audiences

Application Process

After a preferred theme is selected:

1. Read the corresponding theme file from the `themes/` directory
2. Apply the specified colors and fonts consistently throughout the deck
3. Ensure proper contrast and readability
4. Maintain the theme's visual identity across all slides

Create your Own Theme

To handle cases where none of the existing themes work for an artifact, create a custom theme. Based on provided inputs, generate a new theme similar to the ones above. Give the theme a similar name describing what the font/color combinations represent. Use any basic description provided to choose appropriate colors/fonts. After generating the theme, show it for review and verification. Following that, apply the theme as described above.

/web-artifacts-builder

Source: `~/.claude/skills/web-artifacts-builder/SKILL.md`

---

name: web-artifacts-builder description: Suite of tools for creating elaborate, multi-component claude.ai HTML artifacts using modern frontend web technologies (React, Tailwind CSS, shadcn/ui). Use for complex artifacts requiring state management, routing, or shadcn/ui components - not for simple single-file HTML/JSX artifacts. license: Complete terms in LICENSE.txt

Web Artifacts Builder

To build powerful frontend claude.ai artifacts, follow these steps:

1. Initialize the frontend repo using `scripts/init-artifact.sh`
2. Develop your artifact by editing the generated code
3. Bundle all code into a single HTML file using `scripts/bundle-artifact.sh`
4. Display artifact to user

5. (Optional) Test the artifact

Stack: React 18 + TypeScript + Vite + Parcel (bundling) + Tailwind CSS + shadcn/ui

Design & Style Guidelines

VERY IMPORTANT: To avoid what is often referred to as "AI slop", avoid using excessive centered layouts, purple gradients, uniform rounded corners, and Inter font.

Quick Start

Step 1: Initialize Project

Run the initialization script to create a new React project:

```
bash scripts/init-artifact.sh <project-name>
cd <project-name>
```

This creates a fully configured project with:

- □ React + TypeScript (via Vite)
- □ Tailwind CSS 3.4.1 with shadcn/ui theming system
- □ Path aliases (`@/`) configured
- □ 40+ shadcn/ui components pre-installed
- □ All Radix UI dependencies included
- □ Parcel configured for bundling (via .parcelrc)
- □ Node 18+ compatibility (auto-detects and pins Vite version)

Step 2: Develop Your Artifact

To build the artifact, edit the generated files. See **Common Development Tasks** below for guidance.

Step 3: Bundle to Single HTML File

To bundle the React app into a single HTML artifact:

```
bash scripts/bundle-artifact.sh
```

This creates `bundle.html` - a self-contained artifact with all JavaScript, CSS, and dependencies inlined. This file can be directly shared in Claude conversations as an artifact.

Requirements: Your project must have an `index.html` in the root directory.

What the script does:

- Installs bundling dependencies (parcel, @parcel/config-default, parcel-resolver-tspaths, html-inline)
- Creates `.parcelrc` config with path alias support
- Builds with Parcel (no source maps)
- Inlines all assets into single HTML using html-inline

Step 4: Share Artifact with User

Finally, share the bundled HTML file in conversation with the user so they can view it as an artifact.

Step 5: Testing/Visualizing the Artifact (Optional)

Note: This is a completely optional step. Only perform if necessary or requested.

To test/visualize the artifact, use available tools (including other Skills or built-in tools like Playwright or Puppeteer). In general, avoid testing the artifact upfront as it adds latency between the request and when the finished artifact can be seen. Test later, after presenting the artifact, if requested or if issues arise.

Reference

- **shadcn/ui components**: https://ui.shadcn.com/docs/components

/webapp-testing

Source: `~/.claude/skills/webapp-testing/SKILL.md`

---

name: webapp-testing description: Toolkit for interacting with and testing local web applications using Playwright. Supports verifying frontend functionality, debugging UI behavior, capturing browser screenshots, and viewing browser logs. license: Complete terms in LICENSE.txt

Web Application Testing

To test local web applications, write native Python Playwright scripts.

Helper Scripts Available:

- `scripts/with_server.py` - Manages server lifecycle (supports multiple servers)

Always run scripts with `--help` first to see usage. DO NOT read the source until you try running the script first and find that a customized solution is abslutely necessary. These scripts can be very large and thus pollute your context window. They exist to be called directly as black-box scripts rather than ingested into your context window.

Decision Tree: Choosing Your Approach

```
User task → Is it static HTML?
    ├─ Yes → Read HTML file directly to identify selectors
    │         ├─ Success → Write Playwright script using selectors
    │         └─ Fails/Incomplete → Treat as dynamic (below)
    │
    └─ No (dynamic webapp) → Is the server already running?
        ├─ No → Run: python scripts/with_server.py --help
        │        Then use the helper + write simplified Playwright script
        │
        └─ Yes → Reconnaissance-then-action:
            1. Navigate and wait for networkidle
            2. Take screenshot or inspect DOM
            3. Identify selectors from rendered state
            4. Execute actions with discovered selectors
```

Example: Using with_server.py

To start a server, run `--help` first, then use the helper:

Single server:

```
python scripts/with_server.py --server "npm run dev" --port 5173 -- python your_automation.py
```

Multiple servers (e.g., backend + frontend):

```
python scripts/with_server.py \
  --server "cd backend && python server.py" --port 3000 \
  --server "cd frontend && npm run dev" --port 5173 \
  -- python your_automation.py
```

To create an automation script, include only Playwright logic (servers are managed automatically):

```
from playwright.sync_api import sync_playwright
```

```
with sync_playwright() as p:
    browser = p.chromium.launch(headless=True) # Always launch chromium in headless mode
    page = browser.new_page()
    page.goto('http://localhost:5173') # Server already running and ready
    page.wait_for_load_state('networkidle') # CRITICAL: Wait for JS to execute
    # ... your automation logic
    browser.close()
```

Reconnaissance-Then-Action Pattern

1. **Inspect rendered DOM**:

   ```
   page.screenshot(path='/tmp/inspect.png', full_page=True)
   content = page.content()
   page.locator('button').all()
   ```

2. **Identify selectors** from inspection results
3. **Execute actions** using discovered selectors

Common Pitfall

 Don't inspect the DOM before waiting for `networkidle` on dynamic apps **Do** wait for `page.wait_for_load_state('networkidle')` before inspection

Best Practices

- **Use bundled scripts as black boxes** - To accomplish a task, consider whether one of the scripts available in `scripts/` can help. These scripts handle common, complex workflows reliably without cluttering the context window. Use `--help` to see usage, then invoke directly.
- Use `sync_playwright()` for synchronous scripts
- Always close the browser when done
- Use descriptive selectors: `text=`, `role=`, CSS selectors, or IDs
- Add appropriate waits: `page.wait_for_selector()` or `page.wait_for_timeout()`

Reference Files

- **examples/** - Examples showing common patterns:

- `element_discovery.py` - Discovering buttons, links, and inputs on a page
- `static_html_automation.py` - Using file:// URLs for local HTML
- `console_logging.py` - Capturing console logs during automation

Known Issues & Fixes

2026-04-03 16:12:33

Error: Deploy verification was skipped - John claimed 'sve live' based on curl, CEO found 404. Fix: MANDATORY browser click-through test after every deploy before ANY claim to CEO. Add deploy-verify checklist.

Fix: MANDATORY browser click-through test after every deploy before ANY claim to CEO. Add deploy-verify checklist.

/plan-build-test

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

---

name: plan-build-test version: "2.0" level: 3 trigger: "plan-build-test, full-cycle test, playwright, E2E testing, run tests" author: john updated: 2026-03-16 description: Orchestration skill for Plan?Build?Test development cycles. Runs Playwright CLI tests (NOT MCP) against local or remote web apps. Supports E2E testing, visual regression, and mobile viewport testing.

Plan-Build-Test Orchestration Skill

Automates the full development cycle: implement changes → build → test → fix → re-test.

CRITICAL: Playwright CLI ONLY — NEVER use MCP playwright tools. All testing via `npx playwright test` or `./scripts/test-runner.sh`.

Modes

Mode 1: Full Cycle (`\plan-build-test:full-cycle`)

Purpose: Implement feature/fix → build → test → fix failures → visual regression

Agent workflow:

1. **Read requirements**
   - Read task description and acceptance criteria
   - Identify files to change and expected test coverage
2. **Spawn builder subagent**
   - Use Task tool to spawn builder agent with clear file ownership
   - Wait for builder to complete implementation
   - Verify builder marked task as done
3. **Build verification**
   - Run build command: `npx next build` (or relevant for project)
   - Parse output for errors
   - If build fails → analyze errors → spawn builder to fix → re-build
   - Max 3 build iterations before escalating
4. **Start dev server (if testing locally)**
   - If TEST_BASE_URL not set, start dev server: `npx next dev &`
   - Wait for server ready (check http://localhost:3000)
   - If testing remote URL, skip this step
5. **Run E2E tests**
   - Execute: `./scripts/test-runner.sh [--project <project>] [--grep <pattern>]`
   - Parse JSON results from `/tmp/playwright-results.json`
   - Capture:
     - Total tests, passed, failed, skipped
     - Failure details (test title, error message)
     - Screenshot paths from `/tmp/playwright-screenshots/`
6. **Fix failures (if needed)**
   - If tests fail:
     - Analyze failure details and screenshots
     - Identify root cause
     - Spawn builder to fix issues
     - Re-run tests
   - Max 3 fix iterations before escalating

7. **Visual regression (optional)**
   - If changes affect UI:
     - Run: `./scripts/visual-regression.sh`
     - Compare against baseline
     - Report diff percentages
     - Show paths to diff images
   - If no baseline exists:
     - Capture baseline: `./scripts/visual-regression.sh --baseline`
     - Skip comparison (first run)
8. **Report summary**
   - Build status: pass/fail
   - Test results: X/Y passed
   - Failure details (if any) with screenshot references
   - Visual regression status (if run)
   - Next steps or completion confirmation

Variables:

- `{{TASK_DESCRIPTION}}` — What to implement
- `{{PROJECT_DIR}}` — Project root path
- `{{BASE_URL}}` — URL to test (default: http://localhost:3000)
- `{{MAX_ITERATIONS}}` — Max fix attempts (default: 3)

Example usage:

```
\plan-build-test:full-cycle
Task: Implement login form validation
Project: /Users/makinja/ALAI/products/Drop/src/drop-app
Base URL: http://localhost:3000
```

Mode 2: Test Only (`\plan-build-test:test-only`)

Purpose: Run tests against existing deployment (local or remote) without building

Agent workflow:

1. **Accept parameters**
   - URL to test (required, default: http://localhost:3000)
   - Project filter (optional, e.g., "mobile-iphone")
   - Test grep pattern (optional, e.g., "login")
2. **Run tests**

- Execute: `TEST_BASE_URL=<url> ./scripts/test-runner.sh [--project <project>] [--grep <pattern>]`
- Parse results from `/tmp/playwright-results.json`

3. **Report results**
   - Summary: X/Y tests passed
   - If failures:
     - Show failure details (test title + error message)
     - List screenshot paths from `/tmp/playwright-screenshots/`
   - Exit code: 0 = all pass, 1 = failures

Variables:

- `{{BASE_URL}}` — URL to test
- `{{PROJECT}}` — Project filter (optional)
- `{{GREP_PATTERN}}` — Test name filter (optional)

Example usage:

```
\plan-build-test:test-only
URL: https://staging.getdrop.no
Project: mobile-iphone
Pattern: login
```

Mobile testing:

- iPhone viewport: `--project mobile-iphone`
- Galaxy viewport: `--project mobile-galaxy`
- iPad viewport: `--project tablet-ipad`

Mode 3: Visual Check (`\plan-build-test:visual-check`)

Purpose: Capture screenshots and compare against baseline for visual regression detection

Agent workflow:

1. **Check baseline status**
   - Check if baseline exists: `ls tests/visual/baseline/*.png`
   - If no baseline → capture baseline mode
   - If baseline exists → comparison mode
2. **Capture baseline (first run)**
   - Execute: `./scripts/visual-regression.sh --baseline`
   - Saves screenshots to `tests/visual/baseline/`

- Report: "Baseline captured, X screenshots saved"
- Skip comparison (nothing to compare against)

3. **Run comparison (subsequent runs)**
   - Execute: `./scripts/visual-regression.sh [--threshold <percent>]`
   - Default threshold: 5% (customizable)
   - Compares current screenshots vs baseline
   - Generates diff images to `/tmp/visual-diffs/`
4. **Report results**
   - Per-page diff percentages
   - Overall status: pass (no diffs > threshold) or fail (diffs detected)
   - Paths to diff images for review
   - Recommendation: approve new baseline or fix regressions

Variables:

- `{{PROJECT_DIR}}` — Project root path
- `{{THRESHOLD}}` — Max diff percentage allowed (default: 5)

Example usage:

```
\plan-build-test:visual-check
Threshold: 10
```

Workflow:

1. First run: Capture baseline
2. Make UI changes
3. Run visual check → see diffs
4. Review diff images
5. If intentional → update baseline: `./scripts/visual-regression.sh --baseline`
6. If bugs → fix issues → re-run visual check

Key Constraints

1. **Playwright CLI ONLY**
   - NEVER use MCP playwright tools
   - All tests via `npx playwright test` or wrapper scripts
   - No browser automation except through Playwright CLI
2. **URL flexibility**
   - Support local dev: http://localhost:3000
   - Support staging: https://staging.example.com
   - Support production: https://example.com
   - Use TEST_BASE_URL env var to override default

3. **Mobile testing**
   - Use `--project` flag for mobile viewports
   - Available projects: mobile-iphone, mobile-galaxy, tablet-ipad
   - See playwright.config.ts for full project list
4. **JSON results parsing**
   - Always parse `/tmp/playwright-results.json` for structured data
   - Extract: total, passed, failed, skipped, failures[]
   - Reference screenshot paths from `/tmp/playwright-screenshots/`
5. **Screenshot evidence**
   - All failure screenshots saved to `/tmp/playwright-screenshots/`
   - Visual regression diffs saved to `/tmp/visual-diffs/`
   - Include paths in reports for manual review
6. **Iterative fixing**
   - Max 3 iterations for build fixes
   - Max 3 iterations for test fixes
   - After max iterations → escalate to human with detailed failure analysis
7. **Build before test**
   - Full cycle MUST run build before tests
   - Test-only mode assumes build already done
   - Visual check mode can run independently (screenshot capture doesn't require build)

File Locations

- **Test runner:** `./scripts/test-runner.sh`
- **Visual regression:** `./scripts/visual-regression.sh`
- **Playwright config:** `playwright.config.ts`
- **Test results:** `/tmp/playwright-results.json`
- **Screenshots:** `/tmp/playwright-screenshots/`
- **Visual diffs:** `/tmp/visual-diffs/`
- **Visual baseline:** `tests/visual/baseline/`

Example Outputs

Full Cycle Success

```
Task #1234 COMPLETE

Build: ✓ Passed
```

```
Tests: ✓ 15/15 passed
Visual regression: ✓ No changes detected (all diffs < 5%)

Ready for deployment.
```

Full Cycle with Failures

```
Task #1234 — Test failures detected

Build: ✓ Passed
Tests: ✗ 12/15 passed (3 failures)

Failures:
1. "login with valid credentials" — Error: Element not found: button[type="submit"]
   Screenshot: /tmp/playwright-screenshots/login-failure-1.png

2. "dashboard loads after login" — Error: Timeout waiting for selector: h1:has-
text("Dashboard")
   Screenshot: /tmp/playwright-screenshots/dashboard-timeout-2.png

Fix iteration 1/3 in progress...
```

Test Only (Remote)

```
Testing: https://staging.getdrop.no
Project: mobile-iphone
Results: ✓ 8/8 passed

All tests passed on mobile viewport.
```

Visual Check

```
Visual regression results:

✓ login.png — 0.2% diff (PASS)
✓ dashboard.png — 1.8% diff (PASS)
✗ profile.png — 12.5% diff (FAIL — exceeds 5% threshold)
```

```
Review diff: /tmp/visual-diffs/profile-diff.png

Action needed: Review profile page changes or update baseline if intentional.
```

? Operational Limits

- **MAX TURNS:** 30 (build) | 20 (validate) | 10 (lookup)
- Exit cleanly after completing. On 5+ failures: escalate to John with full error context.

/sentinel

Source:

`~/.claude/skills/sentinel/SKILL.md`

---

name: sentinel version: 2.0 description: > Run full system audit using 5-agent team (BA, Architect, Developer, Tester, Validator). Use when: "audit the system", "run sentinel", "system health check", "/sentinel", "review infrastructure", "find issues", "what's broken". argument-hint: "[target] — e.g. 'tools', 'hooks', 'Drop project', 'daemons', or empty for full audit" level: 4 company: ALAI

/sentinel — System Audit Team

Purpose

5-agent parallel audit that delivers a consolidated report with prioritized action items. BA + Architect + Developer + Tester run in parallel → Validator consolidates.

Variables

Variable	Type	Description	Default
`target`	string	Audit scope	full system
`model`	string	Agent model	sonnet
`depth`	string	shallow | deep	deep

Team

Role	Agent	Focus
BA	sentinel-ba.md	Business value, gaps, redundancy, ROI
Architect	sentinel-architect.md	Architecture, integrations, offline/online parity
Developer	sentinel-developer.md	Code quality, dead code, tech debt, bugs
Tester	sentinel-tester.md	Functional testing, daemon health, data integrity
Validator	sentinel-validator.md	Cross-reference, consolidate, final action plan

Workflow

Phase 1: Pre-flight

- Read audit target/scope from $ARGUMENTS
- if no target → set target = "full system"
- if target = "quick" → set depth = shallow (skip code quality, focus on daemons + health)

Phase 2: Parallel Audit (4 agents simultaneously)

Spawn 4 sub-agents in parallel, each with:

1. Role-specific prompt from `~/.claude/agents/sentinel-{role}.md`
2. Audit target
3. Key paths: ~/system/, ~/.claude/, ~/system/databases/

```
[Parallel]:
  Task(sentinel-ba)        → business audit report
  Task(sentinel-architect) → architecture audit report
  Task(sentinel-developer) → code quality report
  Task(sentinel-tester)    → health/functional report
```

Phase 3: Validation (after all 4 complete)

Spawn Validator with all 4 reports as input:

```
Task(sentinel-validator, input=[ba_report, arch_report, dev_report, test_report])
  → consolidated final report
```

Phase 4: Output

- Print final report from Validator
- if critical issues found → create MC tasks via delegate_task
- if minor issues → list as recommendations

Report Format

```
SENTINEL AUDIT REPORT
Target: [scope]
Date: [timestamp]
Model: [sonnet|opus]

CRITICAL (fix immediately):
  [numbered list]

HIGH (fix this week):
  [numbered list]

MEDIUM (backlog):
```

```
  [numbered list]


MC Tasks Created: [list of task IDs]
Next Audit: [recommended interval]
```

$ARGUMENTS

/qa-doc-review

Source: `~/.claude/skills/qa-doc-review/SKILL.md`

---

name: qa-doc-review version: "2.0" level: 3 trigger: "QA review, doc review, documentation review, qa-doc, review documentation, check docs" author: john updated: 2026-03-16

QA-Doc Review — Level 3 Supervised Skill

Sistematski pregled dokumentacije i QA artefakata. Provjerava completeness, accuracy, i linkove.

WHEN TO USE

- IF "doc review", "review docs", "check documentation", "QA doc" → activate
- IF completing a task with docs deliverable → run as post-step validation

WORKFLOW

Step 1: Classify Document Type

```
IF task documentation → validate against GOTCHA acceptance criteria
IF API documentation → check endpoint coverage, examples, error codes
IF runbook/ops doc → check command accuracy (run commands to verify)
IF architecture doc → check against actual system (query HiveMind)
IF changelog → check version format, completeness
```

Step 2: Accuracy Check

```
# For runbooks: verify commands actually work
# Run key commands and compare output to what doc claims
IF command in doc:
  run command → compare output → flag discrepancies
```

Step 3: Quality Checklist

```
[ ] Title and metadata present (date, author, version)
[ ] All claimed commands/URLs are verified working
[ ] No localhost:XXXX references in production docs
[ ] No "TODO" or "FIXME" placeholders left
[ ] Links resolve (internal wiki + external)
[ ] Screenshots/diagrams are current (not stale)
[ ] GOTCHA acceptance criteria met (if task doc)
[ ] HiveMind post confirmed (if knowledge doc)
```

Step 4: BookStack Sync Check

```
# IF doc should be in BookStack:
cat ~/system/config/bookstack-sync-map.json | grep "[filename]"
# Verify sync status
```

OUTPUT FORMAT (report to John, not user)

```
QA-DOC REVIEW REPORT


Status: APPROVED | NEEDS_WORK | BLOCKED
```

```
Document: [title/path]
Type: [task-doc | runbook | api-doc | architecture | changelog]

 BLOCKERS:
  - [issue]

⚠  WARNINGS:
  - [issue]

 CONFIRMED WORKING:
  - [verified items]

BookStack: SYNCED | NOT_SYNCED | N/A
HiveMind: POSTED | NOT_POSTED | N/A

Verdict: [one sentence]
```