How to Choose and Use Claude Skills Effectively

A Practical Guide to Getting the Most from Custom Skills

Table of Contents

Introduction

What Are Claude Skills?

Claude Skills are like **expert guidebooks** that provide:

- Specialized knowledge and workflows
- Best practices from real experience
- Step-by-step methodologies
- Common pitfall prevention
- Reusable patterns and templates

Why Use Skills?

Without Skills:

- ❌ Claude reinvents approaches each time
- ❌ Forgets dependencies
- ❌ Skips testing
- ❌ Makes same mistakes repeatedly

- ❌ You spend hours debugging

With Skills:

- ✅ Consistent, proven approaches
- ✅ Systematic verification
- ✅ Catches issues early
- ✅ First-time success rate higher
- ✅ Save hours of debugging time

The Golden Rule

❌ DON'T Reference ALL Skills Every Time

Why not:

- Wastes context window (limited resource)
- Confuses Claude with irrelevant information
- Slows down processing
- Dilutes focus on the actual task

✅ DO Reference ONLY Relevant Skills

Think of skills like tools in a toolbox:

- You don't bring the whole toolbox to every job
- You pick the RIGHT tool for the specific task
- Sometimes you need multiple tools
- Sometimes just one

Key Insight: More skills ≠ better results. Right skills = better results.

Your Skill Arsenal

Core Skills (Recommended Set)

1. test-driven-implementation

What it does: Systematic 3-phase approach (Plan → Implement with Verification → End-to-End Test)

When to use:

- Building any complex feature

- Feature touches DB + API + UI
- Multi-role functionality
- Data isolation required
- Permission systems

What it prevents:

- Missing database tables/columns
- Untested code
- Data leaks
- Permission bugs
- Integration failures

Think of it as: Your "Process Expert" - ensures systematic approach

---

2. database-migration-management

What it does: Safe schema evolution with rollback capabilities

When to use:

- Creating new tables
- Adding/modifying columns
- Creating indexes
- Adding foreign keys
- Any schema changes

What it prevents:

- Breaking changes
- Lost data
- Missing rollback scripts
- Poorly named migrations
- Unindexed queries

Think of it as: Your "Database Expert" - ensures safe schema changes

---

3. aws-fullstack-deployment

What it does: CDK, Lambda, API Gateway, RDS deployment patterns

When to use:

- Deploying new Lambda functions
- Creating CDK stacks
- Setting up API Gateway routes
- Configuring RDS connections
- VPC/networking setup
- Environment variables

What it prevents:

- Deployment failures
- Configuration errors
- Security misconfigurations
- Resource naming issues
- Missing permissions

Think of it as: Your "DevOps Expert" - ensures proper AWS deployment

---

4. rbac-implementation

What it does: Role-based access control patterns

When to use:

- Adding new user roles
- Implementing permissions
- Admin-only features
- Resource ownership checks
- Role-based filtering

What it prevents:

- Permission bypasses
- Data leaks between roles
- Inconsistent permission checks
- Missing authorization
- Security vulnerabilities

Think of it as: Your "Security Expert" - ensures proper access control

---

5. react-multi-role-ui

What it does: Permission-aware React component patterns

When to use:

- Building role-aware UIs
- Conditional rendering by role
- Admin-only UI elements
- User dashboards
- Permission-based navigation

What it prevents:

- Exposed admin UI to users
- Inconsistent role checks
- Hard-coded permissions
- UI/backend mismatch
- Poor UX for different roles

Think of it as: Your "Frontend Expert" - ensures role-aware UI

---

6. cost-monitoring

What it does: Token tracking and cost analytics patterns

When to use:

- Adding AI features
- Tracking API usage
- Building analytics dashboards
- Cost optimization
- Usage reporting

What it prevents:

- Runaway costs
- Missing usage data
- Poor cost visibility
- Inefficient model usage
- Budget overruns

Think of it as: Your "Finance Expert" - ensures cost visibility

---

Decision Tree

How to Choose Which Skill(s) to Use

```
WHAT AM I BUILDING/CHANGING?

├─ NEW COMPLEX FEATURE (DB + API + UI)?
│  │
│  ├─ Has permissions/roles?
│  │  └─ test-driven-implementation + rbac-implementation
│  │
│  ├─ Has cost tracking?
│  │  └─ test-driven-implementation + cost-monitoring
│  │
│  └─ General feature
│     └─ test-driven-implementation
│
├─ DATABASE CHANGES ONLY?
│  │
│  ├─ Simple column/index?
│  │  └─ database-migration-management
│  │
│  └─ Complex schema changes?
│     └─ test-driven-implementation + database-migration-management
│
├─ AWS DEPLOYMENT/INFRASTRUCTURE?
│  └─ aws-fullstack-deployment
│
├─ UI CHANGES ONLY?
│  │
│  ├─ Role-aware components?
│  │  └─ react-multi-role-ui
│  │
│  └─ Simple UI changes?
│     └─ (No skill needed, or react-multi-role-ui for consistency)
│
├─ PERMISSION/ACCESS CONTROL?
│  └─ test-driven-implementation + rbac-implementation
│
└─ COST TRACKING/ANALYTICS?
   └─ cost-monitoring (+ test-driven if building dashboard)
```

Real-World Examples

Example 1: Building Invitation System (Your Project)

Task: Build complete analyst invitation system

Analysis:

- ✓ New tables needed (user_invitations, session_access)
- ✓ New APIs (create/verify/accept invitation)
- ✓ New UI (invitation modal, signup page)
- ✓ Permissions (admin-only invites)
- ✓ Role-aware UI (hide from analysts)

Skills to Use:

1. test-driven-implementation (overall methodology)
2. database-migration-management (safe migrations)
3. rbac-implementation (admin-only patterns)
4. react-multi-role-ui (signup page, invitation modal)

Prompt:

Build analyst invitation system with token-based signup.

Use these skills:
1. test-driven-implementation - Ensure complete dependency analysis
2. database-migration-management - Create user_invitations and session_access tables
3. rbac-implementation - Admin-only invitation creation
4. react-multi-role-ui - Invitation modal and signup page

Start with test-driven-implementation Phase 1 to list all requirements.

Example 2: Adding is_archived Column

Task: Add `is_archived` boolean to review_sessions table

Analysis:

- ✓ Simple column addition
- ✓ Database change only
- ✓ No API changes
- ✓ No UI changes (yet)

Skills to Use:

1. database-migration-management

Prompt:

Add is_archived boolean column to review_sessions table (default false).

Use: database-migration-management skill

Include index on is_archived for filtering.

Example 3: Deploying Email Notification Lambda

Task: Create Lambda to send email notifications via AWS SES

Analysis:

- ✓ New Lambda function
- ✓ AWS SES integration
- ✓ CDK deployment
- ✓ IAM permissions
- ✓ No database changes

Skills to Use:

1. aws-fullstack-deployment

Prompt:

Create and deploy email notification Lambda function using AWS SES.

Use: aws-fullstack-deployment skill

Requirements:
- Lambda triggered by SQS queue
- Send invitation emails
- Handle SES bounces

Example 4: Building Cost Analytics Dashboard

Task: Admin dashboard showing token usage by user/session

Analysis:

- ✓ New UI component
- ✓ Uses existing token_usage table
- ✓ Admin-only access
- ✓ Needs analytics queries
- ✓ Role-aware dashboard

Skills to Use:

1. cost-monitoring (has SQL queries)
2. react-multi-role-ui (admin-only component)
3. test-driven-implementation (verify data isolation)

Prompt:

Build cost analytics dashboard showing token usage breakdown.

Use these skills:
1. cost-monitoring - Use pre-built analytics queries
2. react-multi-role-ui - Admin-only dashboard component
3. test-driven-implementation - Verify admins see all data, analysts see own

Features needed:
- Total tokens by user
- Cost per session
- Daily usage trends
- Top 10 expensive submissions

Example 5: Adding User Profile Pages

Task: Users can view/edit their profile, admins can view all profiles

Analysis:

- ✓ New table (user_profiles)
- ✓ New API endpoints
- ✓ New UI pages
- ✓ Role-based access (own profile vs all profiles)

- ✓ Data isolation

Skills to Use:

1. test-driven-implementation (full feature)
2. rbac-implementation (permission patterns)
3. react-multi-role-ui (profile components)

Prompt:

Build user profile feature with view/edit capabilities.

Use these skills:
1. test-driven-implementation - Systematic development
2. rbac-implementation - Users see own, admins see all
3. react-multi-role-ui - Role-aware profile pages

Requirements:
- Users: view/edit own profile only
- Admins: view all profiles, edit any profile
- Fields: name, email, bio, avatar

Skill Combination Patterns

Pattern 1: New Full-Stack Feature

test-driven-implementation + [domain skill(s)]

Why: test-driven ensures you don't miss dependencies,
domain skills provide specific patterns

Example:

"Build notifications feature"
→ test-driven-implementation (methodology)
→ aws-fullstack-deployment (if deploying Lambda)

Pattern 2: Database-Only Changes

database-migration-management

Why: Focused on schema changes only

Example:

"Add user_preferences table with foreign key to users"
→ database-migration-management

Pattern 3: Permission/Security Features

test-driven-implementation + rbac-implementation

Why: RBAC provides patterns, test-driven ensures all roles tested

Example:

"Add moderator role with comment deletion powers"
→ test-driven-implementation (test all three roles)
→ rbac-implementation (permission patterns)

Pattern 4: UI-Only Changes

react-multi-role-ui

Why: Focused on frontend patterns

Example:

"Add admin badge to user avatars in header"
→ react-multi-role-ui

Pattern 5: Infrastructure/Deployment

aws-fullstack-deployment

Why: Focused on AWS-specific patterns

Example:

"Increase Lambda memory to 2GB and add CloudWatch alarms"
→ aws-fullstack-deployment

Quick Reference Guide

One-Page Cheat Sheet

What Are You Building?	Skill(s) to Use	Reason
New feature (DB+API+UI)	test-driven-implementation + domain skills	Ensures complete planning
Database changes	database-migration-management	Safe schema evolution
AWS deployment	aws-fullstack-deployment	Proven AWS patterns
Permission system	test-driven + rbac-implementation	Security + testing
Role-aware UI	react-multi-role-ui	Frontend patterns
Cost tracking	cost-monitoring	Usage analytics
Adding new role	test-driven + rbac + react-multi-role	Complete role implementation
Simple column add	database-migration-management	Just schema change
Simple UI tweak	(none or react-multi-role)	Keep it simple

Common Scenarios

Scenario 1: Add Bulk User Invitations

What you're building:

- CSV upload for multiple invitations
- Validation and error handling
- Email sending in batches
- Admin-only feature

Skills:

1. test-driven-implementation
2. rbac-implementation
3. aws-fullstack-deployment (if using SQS for batch processing)

Why:

- test-driven: Ensures you plan CSV parsing, validation, batch processing
- rbac: Admin-only feature
- aws: If using SQS/Lambda for async processing

Scenario 2: Add Email Notifications for Submission Completion

What you're building:

- Trigger email when AI analysis completes
- Email template
- AWS SES setup
- Opt-in/opt-out preferences

Skills:

1. aws-fullstack-deployment
2. test-driven-implementation

Why:

- aws: SES setup, Lambda triggers, email queuing
- test-driven: Ensures you test email delivery, bounce handling

---

Scenario 3: Add Export to Excel Feature

What you're building:

- Export button on submissions list
- Generate Excel with results
- Admin sees all, analysts see own

Skills:

1. react-multi-role-ui
2. rbac-implementation

Why:

- react: Button placement, role-aware rendering
- rbac: Data filtering by role

Scenario 4: Optimize Slow Queries

What you're building:

- Add database indexes
- Rewrite inefficient queries
- Monitor performance

Skills:

1. database-migration-management

Why:

- Just adding indexes, focused database work
- Migration skill has index creation patterns

Scenario 5: Add AI Model Selection (Haiku vs Sonnet)

What you're building:

- New table: model_configurations
- API to select model per session
- UI dropdown for model selection
- Cost tracking by model

Skills:

1. test-driven-implementation
2. cost-monitoring

Why:

- test-driven: New table, API, UI all involved
- cost-monitoring: Track cost difference between models

Prompt Templates

Template 1: Simple Change (One Skill)

```
Task: [Brief description of what to build]

Use: [skill-name] skill

Requirements:
- [Requirement 1]
- [Requirement 2]
- [Requirement 3]

Start by reading the skill, then proceed.
```

Example:

```
Task: Add created_by column to review_sessions table

Use: database-migration-management skill

Requirements:
- Foreign key to users.user_id
- NOT NULL with default to current admin
- Index for filtering

Start by reading the skill, then create migration.
```

Template 2: Complex Feature (Multiple Skills)

```
Task: [Detailed description of feature]

Context: [Why building this, any background]

Use these skills:
1. [primary-skill] - [why needed]
2. [secondary-skill] - [why needed]
3. [tertiary-skill] - [why needed]
```

Requirements:
- [Requirement 1]
- [Requirement 2]
- [Requirement 3]

Start with [primary-skill] Phase 1 analysis.

Example:

Task: Build document collaboration feature where users can leave comments

Context: Users want to discuss submissions without external tools

Use these skills:
1. test-driven-implementation - Ensure complete planning (DB + API + UI)
2. rbac-implementation - Comments visible based on session access
3. react-multi-role-ui - Comment components with role-aware editing

Requirements:
- Comments table (comment_id, submission_id, user_id, text, created_at)
- Users can comment on submissions they have access to
- Users can edit/delete own comments
- Admins can delete any comment
- Real-time update not needed (refresh to see new comments)

Start with test-driven-implementation Phase 1 to list all dependencies.

Template 3: Deployment-Focused

Task: Deploy [component name]

Use: aws-fullstack-deployment skill

Infrastructure needs:
- [Lambda/API Gateway/RDS/etc.]
- [Memory/timeout/concurrency settings]
- [Environment variables]
- [IAM permissions needed]

Start by reading deployment patterns, then create CDK stack.

Example:

```
Task: Deploy image processing Lambda function

Use: aws-fullstack-deployment skill

Infrastructure needs:
- Lambda with 3GB memory (image processing intensive)
- 300 second timeout
- S3 trigger on upload
- Write to RDS with processed metadata
- Environment variables: DB_HOST, DB_USER, DB_PASSWORD, BUCKET_NAME

Start by reading deployment patterns, then create CDK stack.
```

Pro Tips

Tip 1: Default to test-driven for New Features

When in doubt: If the feature touches multiple layers (DB + API + UI), start with `test-driven-implementation`.

Why: It forces you to think through ALL dependencies upfront, preventing the "oh I forgot X table" problem.

Tip 2: Layer Specific Skills on Top

Strategy: Use test-driven as the base process, add domain skills for specifics.

```
test-driven-implementation (base methodology)
  ↓ ensures systematic approach
+ rbac-implementation (if permissions involved)
  ↓ provides permission patterns
+ cost-monitoring (if tracking costs)
  ↓ provides analytics patterns
```

Tip 3: Single Skill for Focused Changes

Don't over-engineer: If you're just adding a column, don't invoke test-driven-implementation.

Examples of single-skill tasks:

- Add database index → database-migration-management
- Deploy Lambda → aws-fullstack-deployment
- Style a button → (no skill needed)
- Add admin badge → react-multi-role-ui

Tip 4: Reference Skills Early in Prompt

Do this:

```
Use test-driven-implementation skill.

Build comments feature with these requirements:
[...]
```

Not this:

```
Build comments feature with these requirements:
[...]

Oh and use test-driven-implementation skill.
```

Why: Claude reads skills BEFORE starting work, not during. Early reference ensures proper approach from the start.

Tip 5: Be Explicit About What You Want

Do this:

```
Use test-driven-implementation skill.

Start with Phase 1 analysis. List ALL database tables, API endpoints, and UI components needed. Get my approval before coding.
```

Not this:

```
Use test-driven-implementation skill and build the feature.
```

Why: Being explicit ensures Claude follows the methodology properly.

Tip 6: Combine Skills for Complex Features

Don't be afraid to use 2-3 skills for complex features.

Example:

Building multi-tenant SaaS?
→ test-driven-implementation (process)
→ database-migration-management (tenant isolation schema)
→ rbac-implementation (tenant-level permissions)
→ aws-fullstack-deployment (multi-region deployment)

But cap at 3-4 skills - beyond that, you're probably trying to do too much at once.

Tip 7: Create Skills for Repetitive Patterns

If you find yourself:

- Giving the same instructions repeatedly
- Fixing the same mistakes repeatedly
- Following the same process repeatedly

→ Create a skill for it!

Troubleshooting

Problem 1: Claude Ignores the Skill

Symptoms:

- Claude doesn't follow methodology
- Skips Phase 1 analysis
- Doesn't reference skill content

Solutions:

1. Be explicit: "Read the test-driven-implementation skill BEFORE starting"
2. Reference skill location: "Located at /mnt/skills/user/test-driven-implementation/SKILL.md"
3. Start with: "Following test-driven-implementation Phase 1..."

Problem 2: Claude Gets Confused with Multiple Skills

Symptoms:

- Mixes up methodologies
- Unclear which skill applies when
- Contradictory approaches

Solutions:

1. Use fewer skills (maximum 3-4)
2. Be explicit about when each applies:

> Use test-driven-implementation for overall process.
> Use rbac-implementation specifically for permission patterns.
> Use react-multi-role-ui specifically for UI components.

3. Prioritize: "Primary skill: test-driven. Secondary: rbac for permission patterns only."

Problem 3: Skill Seems Outdated/Wrong

Symptoms:

- Skill instructions don't match current project structure
- References old patterns
- Misses new tools/libraries

Solutions:

1. Update the skill based on new learnings
2. Add notes: "Note: This project uses X instead of Y mentioned in skill"
3. Create new skill version with updated patterns

Problem 4: Don't Know Which Skill to Use

Symptoms:

- Task seems to fit multiple skills
- Unsure if you need skills at all
- Paralyzed by choice

Solutions:

1. Use decision tree (see above)
2. Default to test-driven for complex features
3. Start with no skill, see if Claude struggles, then add skill
4. Ask yourself: "What's the PRIMARY thing I'm building?" - use that skill

---

Problem 5: Skill Makes Things Slower

Symptoms:

- Simple tasks take longer with skills
- Too much overhead for small changes
- Context window filled with irrelevant info

Solutions:

1. Don't use skills for trivial tasks
2. Use skills selectively (not every task needs one)
3. For simple changes, direct instructions may be faster
4. Skills are for complex/repetitive patterns, not one-offs

Advanced Usage

Creating a Skill Workflow

For major features, create a workflow:

1. **Planning Phase**
   - Use: test-driven-implementation
   - Output: Complete dependency analysis
2. **Database Phase**
   - Use: database-migration-management
   - Output: Migrations with rollback scripts
3. **Backend Phase**
   - Use: rbac-implementation (if needed)
   - Output: API endpoints with permissions
4. **Deployment Phase**
   - Use: aws-fullstack-deployment
   - Output: Deployed Lambda functions
5. **Frontend Phase**
   - Use: react-multi-role-ui
   - Output: Role-aware components
6. **Testing Phase**

- Use: test-driven-implementation Phase 3
- Output: End-to-end test results

Skill Evolution

As you work, your skills should evolve:

Week 1: Basic skill with core patterns
Week 2: Add lessons learned from first feature
Week 3: Add advanced patterns discovered
Month 2: Refine based on multiple features
Month 6: Mature skill with comprehensive patterns

Keep skills living documents!

Conclusion

Key Takeaways

1. **Don't use all skills every time** - Pick the relevant ones
2. **Default to test-driven for complex features** - It ensures systematic approach
3. **Single skill for focused changes** - Don't over-engineer
4. **Skills save time on repetitive patterns** - Use them wisely
5. **Skills evolve with your learning** - Update them regularly

The Mental Model

Think of skills as expert consultants:

- You don't hire all consultants for every project
- You hire the RIGHT consultant for the specific problem
- Sometimes you need multiple consultants
- Sometimes you don't need any

Use skills the same way!

Next Steps

1. **Start simple** - Use one skill at a time
2. **Build experience** - Learn which skills work when

3. **Create more skills** - Capture your patterns
4. **Iterate** - Update skills based on learnings
5. **Share** - Help others avoid your mistakes

Happy building! 🚀

Version: 1.0
Created: February 6, 2026
Based on: Real-world analyst role implementation project
Author: Satnam (with Claude's help)