agent-design-guide

Agent Design Guide

This guide walks you through designing effective custom GitHub Copilot agents from concept to production.

Table of Contents

When to Create an Agent

✅ Good Candidates for Custom Agents

- **Repeated specialized tasks** that you perform weekly or more
- **Structured output requirements** (templates, formats, checklists)
- **Domain-specific analysis** requiring context and expertise
- **Role-based interactions** (architect, tester, product owner)
- **Multi-step workflows** with consistent patterns
- **Quality gates** and review processes

❌ Poor Candidates for Custom Agents

- **One-time tasks** or rare occurrences
- **Simple questions** answerable in standard chat
- **Highly variable tasks** with no consistent pattern
- **Tasks requiring human judgment** without clear criteria
- **General coding assistance** (use standard Copilot)

Decision Criteria

Ask yourself:

1. **Frequency:** Will I use this at least weekly?
2. **Consistency:** Does it follow a repeatable pattern?
3. **Specialization:** Does it require specific expertise or perspective?
4. **Output:** Do I need a consistent, structured format?
5. **Value:** Will it save significant time or improve quality?

If you answered "yes" to 3 or more, consider creating an agent.

Agent Design Process

Step 1: Define the Problem

Questions to answer:

- What task does this agent perform?
- Who is the target user?

- What pain point does it solve?
- What does success look like?

Example:

```
Problem: Developers spend 30+ minutes reviewing code for architectural
compliance, and reviews are inconsistent.

Success: Agent provides consistent architectural review in 5 minutes,
catching 80%+ of common violations.
```

Step 2: Identify the Role

Choose a clear persona:

- Software Architect
- Product Owner
- QA Engineer
- Security Auditor
- Technical Writer
- Domain Expert

Define the perspective:

```
Role: Expert Software Architect specializing in Clean Architecture
and Domain-Driven Design

Perspective: Focus on dependency management, layer boundaries,
and domain modeling patterns
```

Step 3: Define Scope and Boundaries

What the agent DOES:

- Specific tasks it handles
- Types of analysis it performs
- Questions it answers

What the agent DOES NOT do:

- Out-of-scope tasks
- Handoffs to other agents or humans
- Limitations to manage expectations

Example:

```
Scope:
✅ Review code against Clean Architecture principles
✅ Identify dependency violations
✅ Suggest DDD patterns

Out of Scope:
❌ Performance optimization
❌ Security analysis
❌ Implementation of changes
```

Step 4: Design Output Format

Choose structure:

- Sections and headings
- Bullet points or numbered lists
- Code examples
- Priority/severity levels
- Action items

Example structure:

```
# [Title]

## Summary
[High-level assessment]

## Analysis
### [Category 1]
- ✅ Strengths
- ⚠️ Concerns
- ❌ Violations

## Recommendations
1. [Priority 1 - Critical]
2. [Priority 2 - Important]
3. [Priority 3 - Nice to have]
```

Step 5: Write Instructions

See Writing Effective Instructions below.

Step 6: Test with Real Scenarios

See Testing and Iteration below.

Step 7: Document and Share

- Add to Custom Agent Catalog
- Create example scenarios in `docs/requirements/agent-scenarios/`
- Share with team for feedback
- Submit PR following Agent Governance

Agent Anatomy

A custom agent consists of two parts: **Front Matter** (metadata) and **Markdown Body** (instructions).

Front Matter Structure

```yaml
---
name: agent-name
description: One-sentence purpose of the agent
tools: ["read", "list_files", "search"]  # Optional
model: gpt-4o  # Optional, defaults to user's model
---
```

Fields:

Field	Required	Description	Example
`name`	Yes	Kebab-case identifier	`architecture-reviewer`
`description`	Yes	One-sentence summary	`Reviews code for Clean Architecture compliance`
`tools`	No	Tool permissions	`["read", "list_files"]`
`model`	No	Specific AI model	`gpt-4o`

Markdown Body Components

The body defines the agent's behavior through structured instructions:

1. Identity and Role

```
# Identity
You are an expert [role] specializing in [domain].
You have deep knowledge of [technologies/patterns].
```

2. Responsibilities

```
# Responsibilities
- Perform [specific task]
- Analyze [specific aspect]
- Provide [specific output]
- Identify [specific issues]
```

3. Context

```
# Context
This repository follows:
- Clean Architecture (Domain, Application, Infrastructure, API)
- Domain-Driven Design patterns
- TDD with xUnit and FakeItEasy
```

4. Constraints

```
# Constraints
```

```
## ALWAYS
- Use specific examples from the codebase
- Prioritize recommendations by impact
- Reference specific files and line numbers

## NEVER
- Suggest changes outside your expertise
- Make assumptions about requirements
- Provide generic advice without context
```

5. Analysis Process

```
# Analysis Process
1. Review [aspect A]
2. Check for [specific pattern]
3. Identify [specific issues]
4. Validate [specific criteria]
5. Formulate recommendations
```

6. Output Format

```
# Output Format

Provide your analysis in this structure:

## Summary
[2-3 sentences]

## [Section 1]
- ✅ [Positive findings]
- ⚠️ [Concerns]
- ❌ [Violations]

## Recommendations
1. [Action item with rationale]
```

7. Tone and Style (Optional)

```
# Tone
- Professional and constructive
- Educational, not prescriptive
- Focus on "why" not just "what"
```

Writing Effective Instructions

Principles

1. Be Specific, Not Generic

❌ **Bad:** "Review the code for issues"

✅ **Good:** "Review the code for Clean Architecture violations, specifically checking that Domain layer has no dependencies on Infrastructure or Application layers"

2. Use Examples

❌ **Bad:** "Identify DDD patterns"

✅ **Good:**

```
Identify DDD patterns such as:
- Aggregates with factory methods (e.g., Order.Create())
- Value objects with value equality (e.g., Address, Money)
- Strongly-typed IDs (e.g., OrderId, CustomerId)
- Repository interfaces with business-intent names
```

3. Define Success Criteria

❌ **Bad:** "Check if code is good"

✅ **Good:**

```
Code passes if:
✅ Domain layer has zero external dependencies
✅ All aggregates have private constructors
✅ Repository methods use business language, not CRUD
```

4. Provide Structure

❌ **Bad:** "Tell me about the code"

✅ **Good:**

```
Provide analysis in four sections:
1. Layer Analysis (Domain, Application, Infrastructure, API)
2. Dependency Violations (with file references)
3. DDD Pattern Review (Aggregates, Entities, Value Objects)
4. Prioritized Recommendations (Critical, Important, Nice-to-have)
```

5. Use Constraints (ALWAYS/NEVER)

```
## ALWAYS
- Include file paths and line numbers
- Explain the "why" behind each recommendation
- Categorize findings by severity
```

```
## NEVER
- Suggest changes without explaining the benefit
- Use jargon without defining it
- Provide recommendations outside architecture
```

6. Make It Actionable

❌ **Bad:** "Code could be better"

✅ **Good:** "Move `NotificationService` from `Domain/Services/` to `Infrastructure/Services/` because it depends on external email APIs, violating Clean Architecture's dependency rule"

Testing and Iteration

Test-Driven Agent Development

1. **Create test scenarios BEFORE writing instructions**
2. **Define expected output for each scenario**
3. **Test agent against scenarios**
4. **Iterate until output matches expectations**

Sample Test Scenario Template

```
# Test Scenario: [Name]

## Input
[Description of code/context to analyze]

## Expected Output
[Specific format and content expected]

## Success Criteria
- [ ] Output follows defined format
- [ ] Identifies all critical issues
- [ ] Provides actionable recommendations
- [ ] Tone is constructive and educational
```

Testing Workflow

```
1. Write 3-5 test scenarios
   ↓
2. Create initial agent definition
   ↓
3. Test scenario 1
   ↓
4. Review output → Issues found?
   ↓                ↓
   No             Yes → Refine instructions
```

```
   ↓                          ↓
5. Test scenario 2            ↑
   ↓                          |
6. Iterate through all -------+
   ↓
7. Production ready
```

Iteration Checklist

After each test:

- [ ] Does output follow the specified format?
- [ ] Are findings accurate and relevant?
- [ ] Is tone appropriate?
- [ ] Are recommendations actionable?
- [ ] Are examples specific to the codebase?
- [ ] Is anything missing?
- [ ] Is anything extraneous?

Common Refinements

Issue	Solution
Output too verbose	Add word/section limits to instructions
Missing context	Add more examples or constraints
Generic advice	Add "ALWAYS use specific file references"
Inconsistent format	Provide explicit template with placeholders
Wrong focus	Clarify scope and boundaries
Unhelpful tone	Add tone guidelines to instructions

Common Pitfalls

1. Scope Creep

Problem: Agent tries to do too much

Solution: Define narrow, focused role

❌ `"Code Reviewer" (too broad)`
✅ `"Architecture Reviewer" (specific)`

2. Vague Instructions

Problem: Agent provides generic output

Solution: Add specific examples and constraints

```
❌ "Check for best practices"
✅ "Verify all aggregates use private constructors with
   static factory methods (e.g., Order.Create())"
```

3. No Output Format

Problem: Inconsistent, hard-to-use output

Solution: Provide explicit template

```
# Required Output Format

## Summary
[One paragraph]

## Findings
- ✅ [Finding]
- ⚠️ [Finding]
- ❌ [Finding]
```

4. Missing Context

Problem: Agent doesn't understand project conventions

Solution: Add context section

```
# Context
This is a .NET 9 project using:
- Clean Architecture
- xUnit for testing
- Entity Framework Core
- Minimal APIs
```

5. Overly Prescriptive

Problem: Agent dictates instead of advises

Solution: Frame as recommendations

```
❌ "You must change X to Y"
✅ "Consider moving X to Y because [rationale].
   This would improve [specific benefit]."
```

6. No Success Criteria

Problem: Can't measure agent effectiveness

Solution: Define measurable outcomes

```
Success Metrics:
- Identifies 80%+ of architectural violations
- Provides 3-5 actionable recommendations
- Reduces review time from 30 min to 5 min
```

7. Assuming Too Much Knowledge

Problem: Agent uses unexplained jargon

Solution: Define terms or assume beginner audience

```
## Tone
- Explain concepts, don't assume knowledge
- Define acronyms (e.g., "DDD (Domain-Driven Design)")
- Link to documentation when helpful
```

Examples and Templates

Template: Basic Agent Structure

```
---
name: [agent-name]
description: [One-sentence purpose]
tools: ["read", "list_files"]
model: gpt-4o
---

# Identity
You are an expert [role] specializing in [domain].

# Responsibilities
- [Task 1]
- [Task 2]
- [Task 3]

# Context
This project uses:
- [Technology/Pattern 1]
- [Technology/Pattern 2]

# Constraints

## ALWAYS
- [Rule 1]
- [Rule 2]

## NEVER
```

```
- [Rule 1]
- [Rule 2]

# Analysis Process
1. [Step 1]
2. [Step 2]
3. [Step 3]

# Output Format

Provide analysis in this structure:

## Summary
[Description]

## [Section 1]
[Description]

## [Section 2]
[Description]

## Recommendations
[Description]

# Tone
- [Guideline 1]
- [Guideline 2]
```

Example: Migration Planner Agent

```
---
name: migration-planner
description: Plans migration strategies for modernizing legacy code
tools: ["read", "list_files", "search"]
model: gpt-4o
---

# Identity
You are an experienced software architect specializing in legacy
system modernization and incremental migration strategies.

# Responsibilities
- Analyze legacy code and identify migration paths
- Propose incremental, low-risk migration strategies
- Identify dependencies and migration order
- Estimate effort and risk for each migration step
- Suggest strangler fig or anti-corruption layer patterns

# Context
```

```
This repository contains:
- Legacy code in TaskManager.Infrastructure/Legacy/
- Modern code following Clean Architecture
- Migration goal: Move from legacy to modern architecture

# Constraints

## ALWAYS
- Propose incremental migrations, never big-bang rewrites
- Identify risks and mitigation strategies
- Reference specific files and components
- Consider backwards compatibility
- Estimate effort (small/medium/large)

## NEVER
- Suggest rewriting entire systems at once
- Ignore dependencies between components
- Propose migrations without testing strategy
- Assume zero downtime is possible

# Analysis Process
1. Identify legacy components and their dependencies
2. Map legacy components to modern architecture layers
3. Determine migration order based on dependencies
4. Identify strangler fig opportunities
5. Propose testing strategy for each phase
6. Estimate effort and risk

# Output Format

## Migration Plan

### Phase 1: [Name]
**Goal:** [What will be migrated]

**Components:**
- Legacy: [File/class]
- Modern: [Target location]

**Strategy:** [Strangler Fig / Anti-Corruption Layer / Direct Migration]

**Steps:**
1. [Action]
2. [Action]
3. [Action]

**Testing:** [How to validate]

**Effort:** [Small/Medium/Large]
```

```
**Risk:** [Low/Medium/High]

**Dependencies:** [What must be done first]

---

[Repeat for each phase]

## Success Metrics
- [How to measure success]

## Rollback Plan
- [How to revert if needed]

# Tone
- Pragmatic and realistic about effort
- Risk-aware but solution-focused
- Educational about migration patterns
```

Example: Documentation Generator Agent

```
---
name: doc-generator
description: Generates comprehensive documentation from code
tools: ["read", "list_files"]
model: gpt-4o
---

# Identity
You are a technical writer specializing in API and code documentation.

# Responsibilities
- Generate clear, accurate documentation from code
- Explain purpose, parameters, return values, and examples
- Document edge cases and error conditions
- Provide usage examples in context

# Context
Documentation should follow this repository's conventions:
- Use Markdown format
- Include code examples
- Document public APIs
- Explain "why" not just "what"

# Constraints

## ALWAYS
- Include usage examples
```

```
- Document exceptions and error cases
- Use consistent formatting
- Link to related documentation
- Show real code, not pseudocode

## NEVER
- Simply repeat method names as descriptions
- Omit parameter descriptions
- Forget return value documentation
- Skip error handling documentation

# Output Format

# [Method/Class Name]

## Purpose
[What it does and why it exists]

## Signature
```csharp
[Full method signature]
```

Parameters

- `paramName` (Type): [Description]

Returns

[Description of return value]

Exceptions

- `ExceptionType`: [When thrown]

Example

```
[Real usage example]
```

See Also

- [Related methods/classes]

Tone

- Clear and concise
- Educational
- Example-driven
```

Agent Design Checklist

Use this checklist before finalizing your agent:

Planning
- [ ] Problem clearly defined
- [ ] Success criteria established
- [ ] Role and perspective identified
- [ ] Scope and boundaries documented
- [ ] Test scenarios created

Implementation
- [ ] Front matter complete (name, description)
- [ ] Identity and role clearly stated
- [ ] Responsibilities listed
- [ ] Context provided
- [ ] Constraints defined (ALWAYS/NEVER)
- [ ] Analysis process outlined
- [ ] Output format specified with template
- [ ] Tone guidelines included

Testing
- [ ] Tested with all scenarios
- [ ] Output format consistent
- [ ] Findings accurate and relevant
- [ ] Recommendations actionable
- [ ] Tone appropriate
- [ ] Edge cases handled

Documentation
- [ ] Added to Custom Agent Catalog
- [ ] Example scenarios created
- [ ] Usage instructions clear
- [ ] Success metrics defined

Review
- [ ] Peer reviewed by team member
- [ ] Governance process followed
- [ ] Version control committed
- [ ] Team notified of new agent

See Also

- [Custom Agent Catalog](./custom-agent-catalog.md) - All available agents
- [Agent Governance](./agent-governance.md) - Review and versioning process
- [Lab 08: Agent Design](../labs/lab-08-agent-design.md) - Hands-on practice
- [Lab 09: Build Your Own Agent](../labs/lab-09-capstone-build-agent.md) - Capstone exercise
- [Agent Architecture Diagram](../design/diagrams/agent-architecture.md)