custom-agent-catalog

Custom Agent Catalog

This catalog provides a comprehensive reference for all custom GitHub Copilot agents available in this repository.

Available Agents

1. Architecture Reviewer

File: `.github/agents/architecture-reviewer.agent.md`

Purpose: Reviews code for Clean Architecture and Domain-Driven Design (DDD) compliance.

When to Use:

- Before merging feature branches
- During architectural refactoring
- When adding new layers or components

- To validate dependency directions
- For educational feedback on architectural patterns

What It Does:

- Analyzes code structure against Clean Architecture layers
- Identifies dependency violations (e.g., Domain depending on Infrastructure)
- Reviews DDD patterns (aggregates, entities, value objects, repositories)
- Validates bounded contexts and domain modeling
- Provides actionable recommendations with examples

Output Format:

```
# Architecture Review

## Summary
[High-level assessment]

## Layer Analysis
### Domain Layer
- ✅ Strengths
- ⚠️ Concerns

[... for each layer]

## Dependency Analysis
[Violations and recommendations]

## DDD Pattern Review
[Entity, value object, aggregate assessment]

## Recommendations
1. [Prioritized action items]
```

Example Usage:

1. Open relevant files (Domain, Application, Infrastructure)
2. Select "Architecture Reviewer" from agent dropdown
3. Prompt: "Review the Order aggregate and related infrastructure for Clean Architecture compliance"
4. Review structured feedback and prioritize recommendations

Best Practices:

- Provide context by opening related files
- Specify which component or feature to review
- Use early in development to catch issues
- Combine with code review process

---

2. Backlog Generator

File: .github/agents/backlog-generator.agent.md

Purpose: Generates user stories with acceptance criteria following agile best practices.

When to Use:

- Starting a new feature or epic
- Breaking down large requirements
- Planning sprint work
- Converting ideas into actionable stories
- Documenting requirements for the team

What It Does:

- Creates well-formed user stories (As a... I want... So that...)
- Applies INVEST principles (Independent, Negotiable, Valuable, Estimable, Small, Testable)
- Generates clear acceptance criteria
- Identifies dependencies between stories
- Suggests story point estimates
- Proposes story priority based on value

Output Format:

```
# User Stories

## Epic: [Name]

### Story 1: [Title]
**As a** [role]
**I want** [capability]
**So that** [benefit]

**Acceptance Criteria:**
- [ ] Given [context], When [action], Then [outcome]
- [ ] ...

**Dependencies:** [Other stories]
**Estimate:** [Story points]
**Priority:** [High/Medium/Low]

---

[Additional stories...]
```

Example Usage:

1. Select "Backlog Generator" from agent dropdown
2. Prompt: "Generate user stories for a task notification system that alerts users when tasks are assigned or due"
3. Review generated stories
4. Refine acceptance criteria if needed
5. Copy to project management tool

Best Practices:

- Provide context about users and their needs
- Describe the problem, not the solution
- Iterate on generated stories for clarity
- Validate acceptance criteria with stakeholders
- Use for epics and break down into smaller stories

---

3. Test Strategist

File: .github/agents/test-strategist.agent.md

Purpose: Proposes comprehensive test strategies and identifies test scenarios.

When to Use:

- Planning tests for a new feature
- Reviewing test coverage
- Identifying missing test scenarios
- Deciding between unit/integration/e2e tests
- Creating test plans for complex components

What It Does:

- Analyzes code to identify test scenarios
- Categorizes tests (unit, integration, e2e)
- Applies testing pyramid principles
- Suggests test cases for edge cases and error paths
- Proposes test data and fixtures
- Identifies areas needing contract or property-based tests
- Recommends mocking strategies

Output Format:

```
# Test Strategy

## Component: [Name]

## Test Pyramid Distribution
- Unit Tests: [%]
- Integration Tests: [%]
- E2E Tests: [%]

## Unit Tests
### [Class/Method Name]
- **Scenario:** [Description]
- **Given:** [Preconditions]
- **When:** [Action]
- **Then:** [Expected outcome]
- **Type:** Happy path / Edge case / Error case

[Additional scenarios...]
```

```
## Integration Tests
[Scenarios for infrastructure/external dependencies]

## E2E Tests
[User journey scenarios]

## Test Data & Fixtures
[Suggested test data]

## Recommendations
[Prioritized suggestions]
```

Example Usage:

1. Open the code file to test
2. Select "Test Strategist" from agent dropdown
3. Prompt: "Propose a test strategy for the Order aggregate, including unit and integration tests"
4. Review proposed scenarios
5. Implement tests following the strategy
6. Validate coverage

Best Practices:

- Provide context about business rules
- Include domain logic and edge cases
- Use strategy to guide TDD
- Validate suggested scenarios with product owner
- Focus on value, not just coverage percentage

Quick Reference Table

Agent	Primary Use Case	Output Type	Best Stage
Architecture Reviewer	Validate design & dependencies	Structured review	Before merge
Backlog Generator	Create user stories	Story cards	Sprint planning
Test Strategist	Plan test coverage	Test scenarios	Before coding

---

Agent Selection Guide

By Development Phase

```
Requirements → Backlog Generator
   ↓
Design → Architecture Reviewer
   ↓
Testing Plan → Test Strategist
   ↓
Implementation → [Standard Copilot]
   ↓
```

```
Code Review → Architecture Reviewer
   ↓
Test Implementation → Test Strategist
```

By Question Type

Question	Recommended Approach
"How should I structure this feature?"	Architecture Reviewer
"What stories cover this epic?"	Backlog Generator
"What tests do I need?"	Test Strategist
"How do I implement X?"	Standard Copilot Chat
"Explain this code"	Standard Copilot Chat (Ask mode)
"Refactor this method"	Standard Copilot Edit mode

Common Workflows

Workflow 1: New Feature Development

```
1. Backlog Generator
   → Generate user stories from requirements

2. Architecture Reviewer
   → Review proposed design approach

3. Test Strategist
   → Plan test scenarios

4. Standard Copilot (Edit/Chat)
   → Implement code

5. Architecture Reviewer
   → Validate implementation
```

Workflow 2: Refactoring Existing Code

```
1. Architecture Reviewer
   → Identify current issues

2. Test Strategist
   → Ensure test coverage before refactoring

3. Standard Copilot (Edit)
   → Perform refactoring

4. Architecture Reviewer
   → Validate improvements
```

Workflow 3: Sprint Planning

```
1. Backlog Generator
   → Break down epic into stories
```

```
2. Test Strategist
   → Estimate testing effort per story

3. Team Discussion
   → Prioritize and commit to sprint
```

Agent Invocation Examples

Architecture Reviewer Examples

Basic Review:

```
Review the TaskManager.Domain project for Clean Architecture compliance.
```

Focused Review:

```
Analyze the Task aggregate in TaskManager.Domain/Tasks/ for DDD patterns
and dependency management.
```

Refactoring Guidance:

```
I want to move notification logic out of the Task entity. Review the current
design and suggest where this belongs in Clean Architecture.
```

Backlog Generator Examples

From High-Level Requirement:

```
Generate user stories for a task manager where users can create, assign,
and track tasks with due dates and priorities.
```

Breaking Down Epic:

```
Break down the "Task Notifications" epic into user stories. Users should
receive notifications for: task assignments, due dates, and status changes.
```

Adding Details:

```
Enhance these user stories with detailed acceptance criteria:
[paste existing stories]
```

Test Strategist Examples

New Component:

```
Propose a test strategy for the Task aggregate in Domain layer. Include
unit tests for business rules and integration tests for repository.
```

Coverage Analysis:

```
Review test coverage for TaskManager.Application/Services/TaskService.cs
and suggest missing test scenarios.
```

Test Type Guidance:

```
Should I use unit tests or integration tests for validating task
notifications?
What scenarios should each cover?
```

Tips for Effective Agent Use

1. Provide Context

- Open relevant files before invoking agent
- Include design documents in conversation
- Reference related PRs or issues

2. Be Specific

- Name the specific class, method, or component
- Define the scope of review or generation
- State your goals or constraints

3. Iterate

- Review agent output
- Ask follow-up questions
- Refine instructions in subsequent prompts

4. Validate Output

- Don't blindly accept agent recommendations
- Discuss with team for significant decisions
- Use agent output as starting point, not final answer

5. Combine with Other Tools

- Use agents alongside code review
- Integrate into PR process
- Complement with manual testing

Extending the Catalog

Creating Your Own Agent

See Agent Design Guide for detailed instructions.

Quick Checklist:

- [ ] Identify a repeated, specialized task
- [ ]

- [ ] Define clear role and responsibilities
- [ ] Specify output format
- [ ] Add constraints (ALWAYS/NEVER rules)
- [ ] Test with real scenarios
- [ ] Document in this catalog
- [ ] Submit PR for team review

Suggesting Improvements

If you find ways to improve existing agents:

1. Test your proposed changes
2. Document the improvement
3. Update agent definition
4. Update this catalog
5. Submit PR with rationale

Agent Governance

All agents in this catalog follow our Agent Governance process:

- **Versioning:** Changes tracked in git
- **Review:** All agent changes require PR review
- **Testing:** Agents tested with real scenarios before production
- **Documentation:** This catalog updated with each agent change
- **Deprecation:** Outdated agents marked and eventually removed

---

FAQ

Q: Can I use multiple agents in one conversation?
A: You can only have one agent active at a time, but you can invoke different agents in sequence. For parallel reviews, use separate conversations.

Q: What if an agent gives incorrect advice?
A: Agents are tools to assist, not replace, human judgment. Always validate critical decisions. Report issues to improve agent definitions.

Q: Can I modify agents for my needs?
A: Yes! Fork the agent definition, make changes, test thoroughly, and submit a PR if improvements benefit the team.

Q: How do I know which agent to use?
A: See the "Agent Selection Guide" above, or refer to the decision trees in Agent vs Instructions vs Prompts.

Q: Do agents work offline?
A: No, agents require GitHub Copilot service which runs in the cloud.

See Also

- Agent Design Guide - How to create effective agents
- Agent Governance - Versioning and review process
- Lab 06: Introduction to Custom Agents
- Lab 07: Workflow Agents in Action
- Agent Architecture Diagram
- Agent Workflow Patterns