agent-governance

Agent Governance

This document defines the processes and standards for managing custom GitHub Copilot agents in this repository.

Table of Contents

Overview

Purpose

Agent governance ensures that:

- Agents are high-quality and well-tested
- Changes are reviewed and documented
- The agent library remains maintainable
- Team members can trust agent outputs
- Deprecated agents are properly retired

Scope

This governance applies to all files in `.github/agents/` and their associated documentation.

Principles

1. **Quality over Quantity** - Better to have 3 excellent agents than 10 mediocre ones
2. **Test Before Merge** - All agents must be tested with real scenarios
3. **Documented Changes** - Every change requires rationale and examples
4. **Team Ownership** - Agents belong to the team, not individuals
5. **Continuous Improvement** - Regular review and refinement

---

Agent Lifecycle

Stage 1: Proposal

Owner: Anyone on the team

Process:

1. Identify a repeated, specialized task
2. Complete the Agent Proposal Template
3. Share with team for feedback (PR or discussion)
4. Get consensus on value and scope

Exit Criteria:

- [ ] Team agrees agent would add value
- [ ] Scope is clear and focused
- [ ] No overlap with existing agents
- [ ] Success criteria defined

Stage 2: Development

Owner: Proposer + Reviewer

Process:

1. Create agent definition in `.github/agents/[name].agent.md`
2. Create 3-5 test scenarios in `docs/requirements/agent-scenarios/`
3. Test agent against all scenarios
4. Iterate until success criteria met
5. Document in Custom Agent Catalog

Exit Criteria:

- [ ] Agent definition complete
- [ ] All test scenarios pass
- [ ] Catalog updated with usage guide
- [ ] Example invocations documented

Stage 3: Review

Owner: Team reviewer (not the author)

Process:

1. Create PR with agent definition and documentation
2. Reviewer tests agent with scenarios
3. Reviewer validates output quality
4. Address feedback and iterate
5. Get approval from at least one team member

Exit Criteria:

- [ ] Code review approved
- [ ] Test scenarios validated by reviewer
- [ ]

- [ ] Documentation clear and complete
- [ ] No conflicts with existing agents

Stage 4: Production

Owner: Team

Process:

1. Merge PR to main branch
2. Announce new agent to team (Slack, standup, etc.)
3. Monitor usage and feedback
4. Iterate based on real-world use

Exit Criteria:

- [ ] Merged to main
- [ ] Team notified
- [ ] Catalog entry visible
- [ ] Agent appears in VS Code dropdown

Stage 5: Maintenance

Owner: Team (agent champion optional)

Process:

1. Collect feedback from team usage
2. Track issues or improvement ideas
3. Periodic review (quarterly or as needed)
4. Update based on changing needs

Exit Criteria:

- [ ] Feedback addressed
- [ ] Agent remains useful and accurate
- [ ] Documentation up-to-date

Stage 6: Deprecation (if needed)

Owner: Team consensus

See Deprecation Process below.

Change Management

Types of Changes

1. Minor Change (No Review Required)

Examples:

- Typo fixes
- Clarifying wording
- Adding examples
- Formatting improvements

Process:

1. Make change
2. Test with one scenario
3. Commit directly to main or quick PR

2. Moderate Change (Review Required)

Examples:

- Adding new constraints
- Adjusting output format
- Expanding scope slightly
- Improving clarity significantly

Process:

1. Create PR with rationale
2. Test with all scenarios
3. Get one approval
4. Merge

3. Major Change (Full Review Required)

Examples:

- Changing agent role or purpose
- Restructuring output significantly
- Modifying core instructions
- Breaking changes to format

Process:

1. Create PR with detailed rationale
2. Test with all scenarios
3. Get two approvals
4. Update all documentation
5. Announce to team
6. Merge

Change Template

When making changes, include in PR description:

```
## Change Type
[ ] Minor [ ] Moderate [ ] Major

## Rationale
Why is this change needed?

## Impact
What will change for users?

## Testing
- [ ] Tested with scenario 1: [result]
- [ ] Tested with scenario 2: [result]
- [ ] Tested with scenario 3: [result]

## Breaking Changes
[ ] None
[ ] Yes - [describe migration path]

## Documentation Updated
- [ ] Agent definition
- [ ] Custom Agent Catalog
- [ ] Test scenarios
- [ ] Related lab exercises
```

Review Process

Reviewer Checklist

When reviewing an agent change:

Functionality

- [ ] Agent performs stated purpose
- [ ] Output format is consistent
- [ ] Recommendations are actionable
- [ ] Scope is appropriately focused

Quality

- [ ]

- [ ] Instructions are clear and specific
- [ ] Examples provided where helpful
- [ ] Constraints defined (ALWAYS/NEVER)
- [ ] Tone is professional and constructive

Testing

- [ ] Tested with provided scenarios
- [ ] Outputs meet expectations
- [ ] Edge cases considered
- [ ] No hallucinations or inaccuracies

Documentation

- [ ] Catalog entry complete and accurate
- [ ] Example invocations provided
- [ ] Test scenarios included
- [ ] Related docs updated

Integration

- [ ] No conflicts with other agents
- [ ] Follows naming conventions
- [ ] Consistent with team standards
- [ ] Git history clean

Review Timeline

- **Minor changes:** Review within 1 business day
- **Moderate changes:** Review within 2 business days
- **Major changes:** Review within 1 week

Handling Disagreements

If reviewer and author disagree:

1. Discuss rationale in PR comments
2. Test both approaches if feasible
3. Bring to team discussion if unresolved
4. Team decision is final

Versioning Strategy

Version Tracking

Agents don't have explicit version numbers. Instead, we use:

1. **Git History** - Commit messages track changes
2. **Conventional Commits** - Structured commit format
3. **PR Descriptions** - Detailed rationale for changes
4. **Changelog** - Optional for major changes

Commit Message Format

```
<type>(agent): <description>

[optional body]

[optional footer]
```

Types:

- `feat` - New agent or major feature
- `fix` - Bug fix or correction
- `docs` - Documentation only
- `refactor` - Restructuring without behavior change
- `test` - Adding or updating scenarios
- `chore` - Maintenance tasks

Examples:

```
feat(agent): add migration-planner agent

Adds new agent for planning legacy code migrations.
Includes test scenarios and catalog documentation.

fix(architecture-reviewer): correct DDD pattern detection

Repository interface detection was too strict. Now allows
business-intent method names as documented.

docs(backlog-generator): add estimation examples

Added example invocations showing how to request story
point estimates in generated stories.
```

Breaking Changes

If a change breaks existing usage:

1. Mark clearly in commit message:

   ```
   feat(agent)!: restructure output format

   BREAKING CHANGE: Output format now uses sections instead
   of bullet lists. Update any automation that parses output.
   ```

2. Update documentation with migration guide

3. Announce to team with action required

4. Consider deprecation period if major impact

Quality Standards

Agent Must Have

1. **Clear Identity**
    - Well-defined role
    - Specific expertise area
    - Focused scope

2. **Structured Instructions**
    - Responsibilities listed
    - Context provided
    - Constraints defined
    - Process outlined
    - Output format specified

3. **Consistent Output**
    - Follows defined format
    - Predictable structure
    - Actionable content

4. **Documentation**
    - Catalog entry
    - Test scenarios (3-5)
    - Example invocations
    - Usage guidance

5. **Quality Assurance**
    - Tested with real scenarios
    - Reviewed by team member
    - Validated against success criteria

Agent Should Have

1. **Examples in Instructions**
   - Concrete examples of patterns
   - Sample outputs
   - Edge case handling

2. **Tone Guidelines**
   - Professional
   - Constructive
   - Educational

3. **Related Documentation**
   - Links to relevant docs
   - References to standards
   - Connection to workflows

Agent Should Not Have

1. **Generic Advice**
   - "Make code better"
   - "Follow best practices"
   - Without specific criteria

2. **Overlapping Scope**
   - Duplicates another agent
   - Too broad a role
   - Conflicting guidance

3. **Untestable Claims**
   - No way to verify
   - Subjective opinions
   - No success criteria

Deprecation Process

When to Deprecate

Consider deprecation when:

- Agent is rarely used (< 1x per month)
- Functionality merged into another agent
- Original need no longer exists
- Better alternative available
- Agent produces unreliable output

Deprecation Steps

Step 1: Mark as Deprecated

Add to agent front matter:

```
---
name: old-agent
description: [DEPRECATED] Use new-agent instead
deprecated: true
replacement: new-agent
---
```

Add notice to top of instructions:

```
> **⚠️ DEPRECATED**
> This agent is deprecated. Use [new-agent](./new-agent.agent.md) instead.
> This agent will be removed on [date].
```

Step 2: Update Documentation

- Mark as deprecated in Catalog
- Explain migration path
- Reference replacement agent
- Set removal date (30-90 days)

Step 3: Announce Deprecation

- Notify team via Slack/email
- Explain rationale
- Provide migration guidance
- Answer questions

Step 4: Grace Period

- Monitor for any remaining usage
- Help users migrate
- Update any internal automation

Step 5: Remove

After grace period:

- Delete agent file
- Remove from catalog
- Update related documentation
- Commit with clear message

Example commit:

```
chore(agent): remove deprecated old-agent
```

```
Agent deprecated on [date] and replaced by new-agent.
Grace period complete with no remaining usage.

See: [PR link to deprecation]
```

Roles and Responsibilities

Agent Author

Responsibilities:

- Create initial agent definition
- Write test scenarios
- Document in catalog
- Respond to review feedback
- Support initial adoption

Not responsible for:

- Long-term maintenance (team responsibility)
- All future changes
- Solo decision on changes

Agent Reviewer

Responsibilities:

- Test agent with scenarios
- Validate quality and accuracy
- Ensure documentation complete
- Approve or request changes
- Provide constructive feedback

Not responsible for:

- Perfect agents (iteration is expected)
- Rewriting entire agent
- Blocking without rationale

Agent Champion (Optional)

Responsibilities:

- Monitor agent usage and feedback
- Propose improvements
- Triage issues
- Coordinate major changes
- Keep documentation current

Not required:

- Agents can be team-owned without champion
- Championship can rotate

- Multiple champions per agent allowed

Team

Responsibilities:

- Use agents and provide feedback
- Follow governance processes
- Review PRs when assigned
- Maintain quality standards
- Decide on deprecations

Appendix

Agent Proposal Template

```
# Agent Proposal: [Name]

## Problem Statement
What problem does this agent solve?

## Current Workaround
How do we currently handle this?

## Proposed Solution
How will the agent help?

## Role and Expertise
What role will the agent take?

## Success Criteria
How will we measure success?

## Test Scenarios
What scenarios will we test?

## Effort Estimate
How long to develop and test?

## Maintenance Consideration
How much maintenance will this need?

## Team Value
Why is this valuable to the team?
```

Quality Checklist

Before merging a new agent:

- [ ] Agent definition follows template
- [ ] Front matter complete
- [ ] Instructions clear and specific
- [ ] Examples provided
- [ ] Constraints defined
- [ ] Output format specified
- [ ] Tone guidelines included
- [ ] 3-5 test scenarios created
- [ ] All scenarios tested and passing
- [ ] Catalog entry complete
- [ ] Example invocations documented
- [ ] Peer reviewed
- [ ] No overlap with existing agents
- [ ] Follows naming conventions
- [ ] Git commits follow conventions

See Also

- Custom Agent Catalog - All available agents
- Agent Design Guide - How to create agents
- Lab 08: Agent Design
- Lab 09: Build Your Own Agent