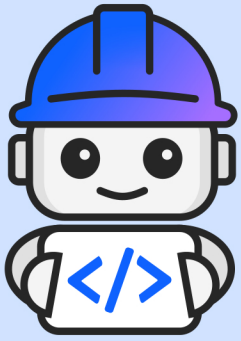


IBM Bob Documentation

Tables of Contents

Welcome to Bob

Bob is an AI-powered coding assistant that helps you write better code faster. Working directly in your editor, Bob understands your codebase and helps you accomplish development tasks with greater efficiency and confidence. Think of Bob as your helpful coding companion—ready to tackle complex problems so you can focus on what matters most: building great software and enjoying the development process along the way.

What you can do with Bob:

Write and modify code

- Generate complete, production-ready code from natural language descriptions
- Refactor existing code to improve readability, performance, or structure
- Debug issues with intelligent error analysis and suggested fixes
- Create new files, functions, and entire projects with proper structure

  Bob's tip: The more specific your request, the better the results. "Create a React component that displays a sortable table of user data" works better than "Make me a table component."

Understand your codebase

Bob helps demystify complex code, turning those "what were they thinking?" moments into clear understanding.

- Get answers to questions about your code's functionality and architecture
- Receive explanations of complex code segments in plain language
- Navigate large codebases more efficiently with contextual understanding
- Learn new frameworks and libraries through interactive guidance

Automate development tasks

Say goodbye to generic commit messages and hello to documentation that actually reflects your code—Bob handles the tedious parts so you can focus on the creative work that actually matters.

- Generate commit messages based on your staged changes
- Create pull request descriptions that clearly explain your changes
- Automate repetitive coding tasks with natural language instructions
- Write and update documentation that stays in sync with your code

Key features

Specialized modes

Bob adapts to your specific needs with purpose-built modes—think of them as different hats Bob wears depending on what you need:

- **Code mode**: Write, modify, and refactor code with precision
- **Ask mode**: Get answers and explanations about your codebase
- **Plan mode**: Plan and design before implementation
- **Advanced mode**: Access extended capabilities, such as MCP, for complex tasks

Each mode optimizes Bob's behavior for different development scenarios—like having the right tool for every job. No need to adjust your communication style when your needs change.

Powerful tools

Bob comes with a comprehensive set of Using tools that extend far beyond simple text generation—these are what make Bob truly special:

- **File operations**: Read, write, and modify files in your project
- **Command execution**: Run commands directly from the chat interface
- **Browser control**: Navigate and interact with web resources
- **Code analysis**: Search codebases and analyze definitions
- **MCP integration**: Connect to external tools and services

These tools work together seamlessly, allowing you to accomplish complex tasks without constantly switching between applications. Less context-switching means more productive coding sessions and fewer interruptions to your flow state.

MCP (Model Context Protocol)

Model Context Protocol (MCP) extends Bob's capabilities by allowing you to add custom tools that supercharge your workflow:

- Connect to databases and APIs
- Access specialized development resources
- Integrate with your organization's internal systems
- Create custom workflows for your specific needs

MCP is available in all modes except Code mode.

Getting started

To get started with Bob:

1. Review the best practices guide to get a brief understanding on how to use Bob effectively. For more information, see Best practices for using Bob.
2. Consult the security guidelines to minimize potential security risks. For more information, see Security guidelines.

   Bob's tip: Start with something you're currently working on. Bob learns best when helping with real problems rather than hypothetical scenarios.

Resources

Ready to explore Bob's capabilities further? Here's where to go next:

- Using modes
- Frequently Asked Questions

Best practices for using Bob

This guide provides essential best practices to help you get the most out of Bob. Whether you're new to Bob or looking to optimize your workflow, these recommendations will help you work more effectively with your AI coding assistant.

Understanding and using modes

Bob's modes are specialized personas that tailor Bob's behavior for specific tasks. Selecting the right mode for each task is crucial for optimal results.

Built-in modes

Bob comes with several built-in modes, each designed for specific purposes:

Mode	Purpose	When to use
Code	Writing and modifying code	For implementing features, fixing bugs, or making code improvements
Advanced	Complex development tasks	When you need full access to all tools, including MCP and browser capabilities
Plan	Planning and designing	Before implementation, when you need to plan architecture or create technical specifications
Ask	Getting information	When you need explanations or information without modifying files

Mode selection strategy

For the most effective workflow:

1. **Start with Plan mode** for new projects or complex features to create a detailed implementation plan before writing any code.
2. **Switch to Code mode** for most day-to-day development tasks when you're ready to implement.
3. **Use Ask mode** when you need explanations or information without modifying files.
4. **Switch to Advanced mode** when you need access to additional tools like browser automation or MCP servers.

For more details on modes and how to switch between them, see Using modes.

Custom modes

You can create specialized modes for specific tasks or workflows:

- Create modes for specialized tasks like security reviews or documentation writing
- Configure custom modes with specific tool access permissions and role definitions
- Share modes with your team to standardize workflows

Learn more about creating and configuring custom modes in the Custom modes documentation.

Workflow optimization

Plan before coding

Always start complex projects or features in Plan mode to:

- Generate a detailed implementation plan
- Break down complex problems into manageable steps
- Identify potential challenges before they arise

- Create a roadmap for implementation

This approach prevents breaking changes and provides a clear direction for development.

Use checkpoints effectively

Checkpoints automatically version your workspace files during Bob tasks, allowing:

- Safe experimentation with AI-suggested changes
- Easy recovery from undesired modifications
- Comparison of different implementation approaches

  Bob's tip: If Bob starts producing subpar work, use the "Restore Files and Task" checkpoint option to roll back to a previous state. This is more effective than trying to correct flawed output with new instructions.

To use a Checkpoint to restore your work to a previous state:

1. Scroll up in your chat interface to locate a Checkpoint you want to restore from.
2. Click the "Restore Checkpoint" button, then click "Restore Files and Task", then "Confirm".

Integrate with version control

Regularly commit and push changes to your version control system:

- Create frequent, small commits with clear messages
- Push changes to remote repositories regularly
- Use branches for experimental features

Git provides an essential safety net to prevent accidental, permanent deletion of code.

Communication strategies

Write effective prompts

The quality of your prompts directly affects the quality of Bob's responses:

- **Be specific and clear**: Vague prompts lead to vague outputs. Detail what Bob should and should not do.
- **Provide examples**: When possible, include examples of the desired output format or style.
- **Use the Enhance Prompt feature**: Click the enhance button to automatically improve your query with additional context and structure.

Learn more about the Enhance Prompt feature in the Enhance prompt documentation.

Use context mentions

Context mentions let you reference specific elements of your project directly in your conversations with Bob:

- Use `@/path/to/file.js` to include specific file contents
- Use `@/path/to/folder` to include all files in a directory
- Use `@problems` to include VS Code Problems panel diagnostics
- Use `@terminal` to include recent terminal output

This approach is more efficient than copying and pasting code or describing file locations.

Learn more about context mentions in the Context mentions documentation.

Manage the context window

To prevent Bob from mixing up your goals:

- Start new tasks regularly with specific aims
- Avoid giving Bob your entire codebase at once
- Use direct file references to provide targeted context
- Break complex tasks into smaller, focused subtasks

Security and control

Configure auto-approval settings

Bob allows you to control which actions require your approval:

- **Manual approval**: Review and approve every action Bob takes (safest setting)
- **Auto-approve**: Grant Bob the ability to run specific tasks without interruption
- **Hybrid approach**: Auto-approve low-risk actions while requiring confirmation for riskier tasks

Configure these settings based on your comfort level and the sensitivity of your project.

Learn more about auto-approval settings in the Auto-approving actions documentation.

Use .bobignore

The `.bobignore` file lets you specify files and directories that Bob should not access or modify:

1. Create a `.bobignore` file in your project root
2. Add patterns for sensitive files, build artifacts, and large assets
3. Use the same syntax as `.gitignore`

This helps protect sensitive information and prevent accidental changes to generated files.

Learn more about controlling file access in the Using .bobignore documentation.

For more detailed information on security guidelines, see Security guidelines.

Set up rules

Bob allows you to specify rules to control how Bob performs tasks:

- Define rules globally or per project in the `.bob` directory
- Create mode-specific rules in `.Bob/rules-{mode-slug}/` directories
- Use rules to enforce coding standards, documentation requirements, or other workflow constraints

For more information on custom modes and rules, see the Custom modes documentation.

Using modes

Bob's modes are specialized personas that tailor Bob's behavior for your specific tasks. Each mode offers different capabilities and access levels to help you accomplish particular goals more efficiently.

Why use different modes?

- **Task specialization:** Get precisely the type of assistance you need for your current task
- **Safety controls:** Prevent unintended file modifications when focusing on planning or learning
- **Focused interactions:** Receive responses optimized for your current activity
- **Workflow optimization:** Seamlessly transition between planning, implementing, debugging, and learning

Switching between modes

You have 4 options to switch modes:

1. **Dropdown menu:** Click the selector to the left of the chat input
2. **Slash command:** Type `/plan`, `/ask`, `/code`, or `/advanced` at the beginning of your message. This will switch to that mode and clear the input field.
3. **Toggle command/Keyboard shortcut:** Use the keyboard shortcut below, applicable to your operating system. Each press cycles through the available modes in sequence, wrapping back to the first mode after reaching the end.

Operating System	Shortcut
macOS	⌘ + .
Windows	Ctrl + .
Linux	Ctrl + .

4. **Accept suggestions:** Click on mode switch suggestions that Bob offers when appropriate

Built-in modes

By default, the following modes are available with Bob:

Code mode

Aspect	Details
Name	`Code`
Role definition	You are Bob, a highly skilled software engineer with extensive knowledge in many programming languages, frameworks, design patterns, and best practices.
Short description	Write, modify, and refactor code.
Tool access	`read`, `edit`, `command`
Use case	General purpose coding tasks, optimized for cost efficiency.

Ask mode

Aspect	Details
Name	`? Ask`
Role definition	You are Bob, a knowledgeable technical assistant focused on answering questions and providing information about software development, technology, and related topics.
Short description	Get answers and explanations.
Tool access	`read`, `browser`, `mcp`
Use case	Conversational questions and information about your code.

Plan mode

Aspect	Details
Name	`Plan`
Role definition	You are Bob, an experienced technical leader who is inquisitive and an excellent planner. Your goal is to gather information and get context to create a detailed plan for accomplishing the user's task, which the user will review and approve before they switch into another mode to implement the solution.
Short description	Plan and design before implementation.
Tool access	`read`, `edit` - markdown only, `browser`, `mcp`
Use case	High-level planning and technical leadership. Big picture thinking!

Advanced mode

Aspect	Details
Name	`Advanced`
Role definition	You are Bob, a highly skilled software engineer with extensive knowledge in many programming languages, frameworks, design patterns, and best practices.
Short description	Advanced version of Code mode, with more tools.
Tool access	All tool groups: `read`, `edit`, `browser`, `command`, `mcp`
Use case	Advanced coding tasks, requiring access to MCP and browser tools. Power users!
Special features	No tool restrictions—full flexibility for all coding tasks

Comparing the built-in modes

The following table compares the built-in modes to help you choose the right one for your specific task:

Mode	Primary Purpose	When to Use	Key Capabilities	Tool Access
Code	Writing and modifying code	When implementing features, fixing bugs, or making code improvements	Efficient code generation, refactoring, and debugging	`read`, `edit`, `command`
? Ask	Getting information and explanations	When you need to understand concepts or analyze existing code without making changes	Detailed explanations, code analysis, and technical recommendations	`read`, `browser`, `mcp`
Plan	Planning and designing	Before implementation, when you need to think about architecture or create a technical plan	System design, breaking down complex problems, creating specifications	`read`, `edit` (markdown only), `browser`, `mcp`
Advanced	Complex development tasks	When you need full access to all tools for sophisticated development workflows	All capabilities of other modes combined with no restrictions	All tool groups

Mode selection guidelines

- **Start with Plan mode** for new projects or complex features to plan before implementation.
- **Use Code mode** for most day-to-day development tasks.
- **Switch to Ask mode** when you need explanations without modifying files.
- **Use Advanced mode** when you need unrestricted access to all tools for complex workflows.

Customizing modes

Tailor Bob Code's behavior by customizing existing modes or creating new specialized assistants. Define tool access, file permissions, and behavior instructions to enforce team standards or create purpose-specific assistants. See Custom Modes documentation for setup instructions.

Customizing modes

You can create **custom modes** to tailor Bob's behavior to specific tasks or workflows. Custom modes can be either **global** (available across all projects) or **project-specific** (defined within a single project).

Why use custom modes

- **Specialization:** Create modes optimized for specific tasks, like "Documentation Writer," "Test Engineer," or "Refactoring Expert."
- **Safety:** Restrict a mode's access to sensitive files or commands. For example, a "Review Mode" could be limited to read-only operations.
- **Experimentation:** Safely experiment with different prompts and configurations without affecting other modes.
- **Team collaboration:** Share custom modes with your team to standardize workflows.

What's included in a custom mode

Custom modes are defined by several key properties. Understanding these concepts will help you tailor Bob's behavior effectively.

UI Field / YAML Property	Conceptual Description
Slug (`slug`)	A **unique internal identifier** for the mode. It's used by Bob to reference the mode, especially for associating mode-specific instruction files.
Name (`name`)	The **display name** for the mode as it appears in the Bob user interface. This should be human-readable and descriptive.
Role Definition (`roleDefinition`)	Defines the **core identity and expertise** of the mode. This text is placed at the beginning of the system prompt. - Its primary function is to define Bob's personality and behavior when this mode is active. - The **first sentence** (up to the first period `.`) serves as a default concise summary for Bob to understand the mode's general purpose. - **However, if the `whenToUse` property is defined, `whenToUse` takes precedence** for summarizing the mode's function, especially in contexts like task orchestration or mode switching. In such cases, the first sentence of `roleDefinition` is less critical for this specific summarization task, though the entire `roleDefinition` is still used when the mode is active to guide its overall behavior.

UI Field / YAML Property	Conceptual Description
Available Tools (`groups`)	Defines the **allowed toolsets and file access permissions** for the mode. - In the UI, this corresponds to selecting which general categories of tools (like reading files, editing files, browsing, or executing commands) the mode can use. - File type restrictions for the "edit" group are typically managed via manual YAML/JSON configuration or by asking Bob to set them up, as detailed in the Property details for `groups`.

| Custom Instructions (optional) (`customInstructions`) | **Specific behavioral guidelines** or rules for the mode.
- These instructions are added near the end of the system prompt to further refine Bob's behavior beyond the `roleDefinition`.
- This can be provided directly in the configuration or via separate instruction files. |

Methods for creating and configuring custom modes

You can create and configure custom modes in several ways:

Ask Bob! (Recommended)

You can quickly create a basic custom mode by asking Bob to do it for you. For example:

```
Create a new mode called "Documentation Writer". It should only be able to read files and write Markdown files.
```

Bob will guide you through the process, prompting for necessary information for the properties described in the What's included in a custom mode table. Bob will create the mode using the preferred YAML format. For fine-tuning or making specific adjustments later, you can use the Prompts tab or manual configuration.

Using the Prompts tab

1. **Open Prompts Tab:** Click the icon in the Bob top menu bar.
2. **Create New Mode:** Click the button to the right of the Modes heading.
3. **Fill in Fields:**

The interface provides fields for `Name`, `Slug`, `Save Location`, `Role Definition`, `When to Use (optional)`, `Available Tools`, and `Custom Instructions`. After filling these, click the "Create Mode" button. Bob will save the new mode in YAML format.

Refer to the What's included in a custom mode table for conceptual explanations of each property. File type restrictions for the "edit" tool group can be added by asking Bob or through manual YAML/JSON configuration.

3. Manual configuration (YAML & JSON)

You can directly edit the configuration files to create or modify custom modes. This method offers the most control over all properties. Bob now supports both YAML (preferred) and JSON formats.

- **Global Modes:** Edit the `custom_modes.yaml` (preferred) or `custom_modes.json` file. Access it via **Prompts Tab** > (Settings Menu icon next to "Global Prompts") > "Edit Global Modes".
- **Project Modes:** Edit the `.Bobmodes` file (which can be YAML or JSON) in your project root. Access it via **Prompts Tab** > (Settings Menu icon next to "Project Prompts") > "Edit Project Modes".

These files define an array/list of custom modes.

YAML Example (`custom_modes.yaml` or `.Bobmodes`):

```yaml
customModes:
  - slug: docs-writer
    name:   Documentation Writer
    roleDefinition: You are a technical writer specializing in clear
documentation.
    whenToUse: Use this mode for writing and editing documentation.
    customInstructions: Focus on clarity and completeness in documentation.
    groups:
      - read
      - - edit # This group allows editing specific files
        - fileRegex: \.(md|mdx)$ # Regex for Markdown files
          description: Markdown files only
      - browser
  - slug: another-mode
    name: Another Mode
    # ... other properties
```

JSON Alternative (`custom_modes.json` or `.Bobmodes`):

```json
{
  "customModes": [
    {
      "slug": "docs-writer",
      "name": "  Documentation Writer",
      "roleDefinition": "You are a technical writer specializing in clear
documentation.",
      "whenToUse": "Use this mode for writing and editing documentation.",
      "customInstructions": "Focus on clarity and completeness in documentation.",
      "groups": [
        "read",
        ["edit", { "fileRegex": "\\.(md|mdx)$", "description": "Markdown files
only" }],
        "browser"
      ]
    },
    {
      "slug": "another-mode",
      "name": "Another Mode"
    }
  ]
}
```

YAML/JSON property details

`slug`

- **Purpose:** A unique identifier for the mode.
- **Format:** Use lowercase letters, numbers, and hyphens.
- **Usage:** Used internally and in file/directory names for mode-specific rules (e.g., `.Bob/rules-{slug}/`).
- **Recommendation:** Keep it short and descriptive.
- *YAML Example:* `slug: docs-writer`
- *JSON Example:* `"slug": "docs-writer"`

`name`

- **Purpose:** The display name shown in the Bob UI.
- **Format:** Can include spaces and proper capitalization.
- *YAML Example:* `name:   Documentation Writer`
- *JSON Example:* `"name": "Documentation Writer"`

`roleDefinition`

- **Purpose:** Detailed description of the mode's role, expertise, and personality.
- **Placement:** This text is placed at the beginning of the system prompt when the mode is active.
- **Important First Sentence:** The first sentence (up to the first period `.`) serves as a default concise summary for Bob to understand the mode's general purpose. **However, if the `whenToUse` property is defined, `whenToUse` takes precedence** for summarizing the mode's function, especially in contexts like task orchestration or mode selection.
- *YAML Example (multi-line):*

```
roleDefinition: >-
  You are a test engineer with expertise in:
  - Writing comprehensive test suites
  - Test-driven development
```

- *JSON Example:* `"roleDefinition": "You are a technical writer specializing in clear documentation."`

`groups`

- **Purpose:** Array/list defining which tool groups the mode can access and any file restrictions.
- **Available Tool Groups (Strings):** `"read"`, `"edit"`, `"browser"`, `"command"`, `"mcp"`.
- **File Restrictions for "edit" group:**
  - To apply file restrictions, the "edit" entry becomes a list (YAML) or array (JSON) where the first element is `"edit"` and the second is a map/object defining the restrictions.
  - `fileRegex`: A regular expression string to control which files the mode can edit.
    - In YAML, typically use single backslashes for regex special characters (e.g., `\.md$`).
    - In JSON, backslashes must be double-escaped (e.g., `\\.md$`).
  - `description`: An optional string describing the restriction.
  - For more complex patterns, see Understanding regex in custom modes.
- *YAML Example:*

```
groups:
  - read
  - - edit # Start of "edit" tool with restrictions
    - fileRegex: \.(js|ts)$ # Restriction map for JS/TS files
      description: JS/TS files only
  - command
```

- *JSON Example:*

```
"groups": [
  "read",
  ["edit", { "fileRegex": "\\.(js|ts)$", "description": "JS/TS files only"
}],
  "command"
]
```

`whenToUse`

- **Purpose:** (Optional) Provides guidance for Bob to understand what this mode does. Used by the Orchestrator mode and for mode switching.
- **Format:** A string describing ideal scenarios or task types for this mode.
- **Usage:** If populated, Bob uses this description. Otherwise, the first sentence of `roleDefinition` is used.
- *YAML Example:* `whenToUse: This mode is best for refactoring Python code.`
- *JSON Example:* `"whenToUse": "This mode is best for refactoring Python code."`

`customInstructions`

- **Purpose:** A string containing additional behavioral guidelines for the mode.
- **Placement:** This text is added near the end of the system prompt.
- **Supplementing:** Can be supplemented by Mode-specific instructions via files/directories.
- *YAML Example (multi-line):*

```
customInstructions: |-
  When writing tests:
  - Use describe/it blocks
  - Include meaningful descriptions
```

- *JSON Example:* `"customInstructions": "Focus on explaining concepts and providing examples."`

Benefits of YAML format

YAML is now the preferred format for defining custom modes due to several advantages over JSON:

- **Readability:** YAML's indentation-based structure is often easier for humans to read and understand complex configurations.
- **Comments:** YAML allows for comments (lines starting with `#`), making it possible to annotate your mode definitions.

```
customModes:
  - slug: security-review
    name:   Security Reviewer
    # This mode is restricted to read-only access
    roleDefinition: You are a security specialist reviewing code for
vulnerabilities.
    whenToUse: Use for security reviews and vulnerability assessments.
    # Only allow reading files, no editing permissions
    groups:
      - read
      - browser
```

- **Multi-line Strings:** YAML provides cleaner syntax for multi-line strings (e.g., for `roleDefinition` or `customInstructions`) using `|` (literal block) or `>` (folded block).

```
customModes:
  - slug: test-engineer
    name:   Test Engineer
    roleDefinition: >-
      You are a test engineer with expertise in:
      - Writing comprehensive test suites
      - Test-driven development
      - Integration testing
      - Performance testing
    customInstructions: |-
      When writing tests:
      - Use describe/it blocks
      - Include meaningful descriptions
      - Test edge cases
      - Ensure proper coverage
    # ... other properties
```

- **Less Punctuation:** YAML generally requires less punctuation (like commas and braces) compared to JSON, reducing syntax errors.
- **Editor Support:** Most modern code editors provide excellent syntax highlighting and validation for YAML files, further enhancing readability and reducing errors.

While JSON is still fully supported, new modes created via the UI or by asking Bob will default to YAML.

Tips for working with YAML

When editing YAML manually, keep these points in mind:

- **Indentation is Key:** YAML uses indentation (spaces, not tabs) to define structure. Incorrect indentation is the most common source of errors. Ensure consistent spacing for nested elements.
- **Colons for Key-Value Pairs:** Keys must be followed by a colon and a space (e.g., `slug: my-mode`).
- **Hyphens for List Items:** List items start with a hyphen and a space (e.g., `- read`).
- **Validate Your YAML:** If you encounter issues, use an online YAML validator or your editor's built-in validation to check for syntax errors.

Migration to YAML format

- **Global Modes:** The migration from `custom_modes.json` to `custom_modes.yaml` happens automatically when Bob starts up, under these conditions:

    1. Bob starts up.
    2. A `custom_modes.json` file exists.
    3. No `custom_modes.yaml` file exists yet. The migration process reads the existing JSON file, converts it to YAML format, creates a new `custom_modes.yaml` file, and preserves the original JSON file (e.g., by renaming it) for rollback purposes. If `custom_modes.yaml` already exists, it will be used, and no automatic migration of `custom_modes.json` will occur.
- **Project Modes (`.Bobmodes`):**

    - **No automatic startup migration:** Unlike global modes, project-specific `.Bobmodes` files are not automatically converted from JSON to YAML simply when Bob starts.
    - **Format Detection:** Bob can read `.Bobmodes` files in either YAML or JSON format.
    - **Conversion on UI Edit:** If you edit a project-specific mode through the Bob UI (e.g., via the Prompts Tab), and the existing `.Bobmodes` file is in JSON format, Bob will save the changes in YAML format. This effectively converts the file to YAML. The original JSON content will be overwritten with YAML.
    - **Manual Conversion:** If you want to convert an existing `.Bobmodes` JSON file to YAML without making UI edits, you'll need to do this manually. You can:
        1. Open your existing JSON `.Bobmodes` file.
        2. Convert its content to YAML (you can ask Bob to help with this, or use an online converter).
        3. Replace the content of your `.Bobmodes` file with the new YAML content, or rename the old file (e.g., `.Bobmodes.json.bak`) and save the new content into a file named `.Bobmodes`. Ensure the resulting YAML is valid.

For manual conversions of `.Bobmodes` files, you can use online JSON to YAML converters or ask Bob to help reformat a specific mode configuration from JSON to YAML. Always validate your YAML before saving.

Mode-specific instructions via files/directories

You can provide instructions for custom modes using dedicated files or directories within your workspace. This allows for better organization and version control compared to only using the `customInstructions` property.

Preferred Method: Directory (`.Bob/rules-{mode-slug}/`)

```
.
├── .Bob/
```

```
│   └── rules-docs-writer/  # Example for mode slug "docs-writer"
│       ├── 01-style-guide.md
│       └── 02-formatting.txt
└── ... (other project files)
```

Fallback Method: Single File (`.bobrules-{mode-slug}`)

```
.
├── .bobrules-docs-writer  # Example for mode slug "docs-writer"
└── ... (other project files)
```

The directory method takes precedence if it exists and contains files.

In addition to the **`customInstructions`** property, you can provide mode-specific instructions via files in your workspace. This is particularly useful for:

- Organizing lengthy or complex instructions into multiple, manageable files.
- Managing instructions easily with version control.
- Allowing non-technical team members to modify instructions without editing YAML/JSON.

There are two ways Bob loads these instructions, with a clear preference for the newer directory-based method:

1. Preferred Method: Directory-Based Instructions (`.Bob/rules-{mode-slug}/`)

- **Structure:** Create a directory named `.Bob/rules-{mode-slug}/` in your workspace root. Replace `{mode-slug}` with your mode's slug (e.g., `.Bob/rules-docs-writer/`).
- **Content:** Place one or more files (e.g., `.md`, `.txt`) containing your instructions inside this directory. You can organize instructions further using subdirectories; Bob reads files recursively, appending their content to the system prompt in **alphabetical order** based on filename.
- **Loading:** All instruction files found within this directory structure will be loaded and applied to the specified mode.

2. Fallback (Backward Compatibility): File-Based Instructions (`.bobrules-{mode-slug}`)

- **Structure:** If the `.Bob/rules-{mode-slug}/` directory **does not exist or is empty**, Bob will look for a single file named `.bobrules-{mode-slug}` in your workspace root (e.g., `.bobrules-docs-writer`).
- **Loading:** If found, the content of this single file will be loaded as instructions for the mode.

Precedence:

- The **directory-based method (`.Bob/rules-{mode-slug}/`) takes precedence**. If this directory exists and contains files, any corresponding root-level `.bobrules-{mode-slug}` file will be **ignored** for that mode.
- This ensures that projects migrated to the new directory structure behave predictably, while older projects using the single-file method remain compatible.

Combining with `customInstructions`:

- Instructions loaded from either the directory or the fallback file are combined with the `customInstructions` property defined in the mode's configuration.
- Typically, the content from the files/directories is appended after the content from the `customInstructions` property.

Configuration precedence

Mode configurations are applied in this order:

1. Project-level mode configurations (from `.Bobmodes` - YAML or JSON)
2. Global mode configurations (from `custom_modes.yaml`, then `custom_modes.json` if YAML not found)
3. Default mode configurations

This means that project-specific configurations will override global configurations, which in turn override default configurations. You can override any default mode by including a mode with the same slug in your global or project-specific configuration.

- **Note on Instruction Files:** Within the loading of mode-specific instructions from the filesystem, the directory `.Bob/rules-{mode-slug}/` takes precedence over the single file `.bobrules-{mode-slug}` found in the workspace root.

Overriding default modes

You can override Bob's built-in modes (like `Code`, `? Ask`, `Plan`, with customized versions. This is done by creating a custom mode with the same slug as a default mode (e.g. `code`).

Overriding modes globally

To customize a default mode across all your projects:

1. **Open Prompts Tab:** Click the icon.
2. **Access Settings Menu:** Click the icon next to "Global Prompts".
3. **Edit Global Modes:** Select "Edit Global Modes" to edit `custom_modes.yaml` (or `custom_modes.json`).
4. **Add Your Override:**

YAML Example:

```
customModes:
  - slug: code # Matches the default 'code' mode slug
    name:   Code (Global Override) # Custom display name
    roleDefinition: You are a software engineer with global-specific constraints.
    whenToUse: This globally overridden code mode is for JS/TS tasks.
    customInstructions: Focus on project-specific JS/TS development.
    groups:
      - read
      - - edit
        - fileRegex: \.(js|ts)$
          description: JS/TS files only
```

JSON Alternative:

```
{
  "customModes": [{
    "slug": "code",
    "name": "  Code (Global Override)",
    "roleDefinition": "You are a software engineer with global-specific
constraints",
    "whenToUse": "This globally overridden code mode is for JS/TS tasks.",
    "customInstructions": "Focus on project-specific JS/TS development",
    "groups": [
      "read",
      ["edit", { "fileRegex": "\\.(js|ts)$", "description": "JS/TS files only" }]
    ]
```

```
  }]
}
```

This example replaces the default `Code` mode with a version restricted to JavaScript and TypeScript files.

Project-specific mode override

To override a default mode for just one project:

1. **Open Prompts Tab:** Click the icon.
2. **Access Settings Menu:** Click the icon next to "Project Prompts".
3. **Edit Project Modes:** Select "Edit Project Modes" to edit the `.Bobmodes` file (YAML or JSON).
4. **Add Your Override:**

YAML Example:

```
customModes:
  - slug: code # Matches the default 'code' mode slug
    name:   Code (Project-Specific) # Custom display name
    roleDefinition: You are a software engineer with project-specific constraints
for this project.
    whenToUse: This project-specific code mode is for Python tasks within this
project.
    customInstructions: Adhere to PEP8 and use type hints.
    groups:
      - read
      - - edit
        - fileRegex: \.py$
          description: Python files only
      - command
```

JSON Alternative:

```
{
  "customModes": [{
    "slug": "code",
    "name": "  Code (Project-Specific)",
    "roleDefinition": "You are a software engineer with project-specific
constraints for this project.",
    "whenToUse": "This project-specific code mode is for Python tasks within this
project.",
    "customInstructions": "Adhere to PEP8 and use type hints.",
    "groups": [
      "read",
      ["edit", { "fileRegex": "\\.py$", "description": "Python files only" }],
      "command"
    ]
  }]
}
```

Project-specific overrides take precedence over global overrides.

Common use cases for overriding default modes

- **Restricting file access:** Limit a mode to specific file types.
- **Specializing behavior:** Customize expertise for your tech stack.
- **Adding custom instructions:** Integrate project standards.
- **Changing available tools:** Remove tools to prevent unwanted operations.

When overriding default modes, test carefully. Consider backing up configurations before major changes.

Understanding regex in custom modes

Regular expressions (`fileRegex`) offer fine-grained control over file editing permissions.

Let Bob build your regex patterns

Instead of writing complex regex manually, ask Bob:

```
Create a regex pattern that matches JavaScript files but excludes test files
```

Bob will generate the pattern. Remember to adapt it for YAML (usually single backslashes) or JSON (double backslashes).

When you specify `fileRegex`, you're creating a pattern that file paths must match.

Important rules for `fileRegex`:

- **Escaping in JSON:** In JSON strings, backslashes (\) must be double-escaped (e.g., `\\.md$`).
- **Escaping in YAML:** In unquoted or single-quoted YAML strings, a single backslash is usually sufficient for regex special characters (e.g., `\.md$`).
- **Path Matching:** Patterns match against the full relative file path from your workspace root (e.g., `src/components/button.js`).
- **Case Sensitivity:** Regex patterns are case-sensitive by default.

Common pattern examples: (Note: JSON examples show double backslashes; YAML would typically use single backslashes for these.)

Pattern (Conceptual / YAML-like)	JSON `fileRegex` Value	Matches	Doesn't Match
`.md$`	`"\.md$"`	`readme.md`, `docs/guide.md`	`script.js`, `readme.md.bak`
`^src/.*`	`"^src/.*"`	`src/app.js`, `src/components/button.tsx`	`lib/utils.js`, `test/src/mock.js`
`.(css	scss)$`	`".(css	scss)$"`
`docs/.*.md$`	`"docs/.*\.md$"`	`docs/guide.md`, `docs/api/reference.md`	`guide.md`, `src/docs/notes.md`
`^(?!.*(test|spec)).(js|ts)$`	`"^(?!.*(test|spec))\.(js|ts)$"`	`app.js`, `utils.ts`	`app.test.js`, `utils.spec.js`, `app.jsx`

Key regex building blocks:

- `\.`: Matches a literal dot. (YAML: `\.`, JSON: `\\.`)
- `$`: Matches the end of the string.
- `^`: Matches the beginning of the string.
- `.*`: Matches any character (except newline) zero or more times.
- `(a|b)`: Matches either "a" or "b". (e.g., `\.(js|ts)$`)
- `(?!...)`: Negative lookahead.

Testing your patterns:

1. Test on sample file paths. Online regex testers are helpful.
2. Remember the escaping rules for JSON vs. YAML.
3. Start simple and build complexity.

Custom instructions

Custom instructions shape how Bob responds to your requests, aligning output with your specific preferences and project requirements. This feature gives you control over Bob's coding style, documentation approach, and decision-making processes.

Understanding custom instructions

Custom instructions extend beyond Bob's default behavior by defining specific preferences, constraints, and guidelines. These instructions direct Bob on how to approach your tasks in ways that match your exact needs.

You can specify:

- Coding style preferences
- Documentation formats and standards
- Testing methodologies and requirements
- Project workflows and processes
- Team-specific conventions and practices

Instruction configuration options

Bob offers multiple ways to configure custom instructions, each with different scopes and priority levels.

Global and workspace scopes

Bob recognizes two primary instruction scopes:

1. **Global instructions**: Apply automatically across all your projects
2. **Workspace instructions**: Apply only within your current project

Directory organization

You can organize custom instructions using either directory structures or single files:

Global rules directories

```
# Linux/macOS
~/.bob/rules/                  # General rules for all modes
~/.bob/rules-{modeSlug}/       # Mode-specific rules (e.g., rules-code/)

# Windows
%USERPROFILE%\.bob\rules\
%USERPROFILE%\.bob\rules-{modeSlug}\
```

Workspace rules

Preferred method: Directory-based

```
.
├── .bob/
│   ├── rules/                 # Workspace-wide rules
│   │   ├── 01-general.md
│   │   └── 02-coding-style.txt
│   └── rules-code/            # Rules for "code" mode
```

```
│       ├── 01-js-style.md
│       └── 02-ts-style.md
└── ... (other project files)
```

Fallback method: File-based

```
.
├── .bobrules               # Workspace-wide rules (single file)
├── .bobrules-code          # Rules for "code" mode (single file)
└── ... (other project files)
```

Configuring custom instructions

Setting up global instructions

Configure global instructions that apply across all workspaces:

1. Click the icon in the Bob top menu bar to open the Prompts tab
2. Locate the "Custom Instructions for All Modes" section
3. Enter your instructions in the provided text area
4. Click "Done" to save your changes

Creating a global rules directory

Global rules directories provide reusable instructions that automatically apply to all your projects.

Advantages of global rules

Without global rules:

- Identical rules must be copied to each new project
- Updates require manual changes across multiple projects
- Projects lack consistency in AI behavior

With global rules:

- Define your preferred standards once
- Override specific rules per project as needed
- Maintain consistency across all projects
- Update all projects simultaneously

Global rules setup process

1. Create the global rules directory:

```
# Linux/macOS
mkdir -p ~/.bob/rules

# Windows
mkdir %USERPROFILE%\.bob\rules
```

2. Add general rules (~/.bob/rules/coding-standards.md):

```
# Global coding standards
1. Always use TypeScript for new projects
2. Write unit tests for all new functions
3. Use descriptive variable names
4. Add JSDoc comments for public APIs
```

3. Add mode-specific rules (~/.bob/rules-code/typescript-rules.md):

```
# TypeScript code mode rules
1. Use strict mode in tsconfig.json
2. Prefer interfaces over type aliases for object shapes
3. Always specify return types for functions
```

Available rule directories

Directory	Purpose
`rules/`	General rules applied to all modes
`rules-code/`	Rules specific to Code mode
`rules-Plan/`	Rules for system architecture tasks
`rules-{mode}/`	Rules for any custom mode

Configuring workspace-level instructions

Workspace-level instructions apply only within your current project.

Using files and directories for workspace instructions

Preferred method: Directory-based (.bob/rules/)

1. Create a `.bob/rules/` directory in your workspace root
2. Add instruction files (e.g., .md, .txt) inside this directory
3. Bob reads all files recursively, including subdirectories
4. Files are processed in alphabetical order by filename

   Note: When the `.bob/rules/` directory exists but contains no files, Bob falls back to the `.bobrules` file.

Fallback method: File-based (.bobrules)

If `.bob/rules/` doesn't exist or contains no files, Bob looks for a single `.bobrules` file in the workspace root.

Mode-specific instructions

Configure mode-specific instructions using either of these methods:

1. Through the Prompts tab:

1. Click the icon in the Bob top menu bar to open the Prompts tab
2. Under the Modes heading, click the button for your target mode
3. Enter instructions in the "Mode-specific Custom Instructions (optional)" field
4. Click "Done" to save your changes

   Note: When a mode is global (not workspace-specific), any custom instructions for that mode apply globally across all workspaces.

2. Using rule files or directories:

Preferred method: Directory-based (.bob/rules-{modeSlug}/)

1. Create a `.bob/rules-{modeSlug}/` directory (e.g., `.bob/rules-code/`) in your workspace root
2. Add instruction files inside this directory

3. Files are read recursively and processed in alphabetical order by filename

Fallback method: File-based (.bobrules-{modeSlug})

If `.bob/rules-{modeSlug}/` doesn't exist or contains no files, Bob looks for a single `.bobrules-{modeSlug}` file (e.g., `.bobrules-code`) in the workspace root.

Instruction priority and combination

Bob combines instructions from multiple sources in this specific order:

1. Global rules (from ~/.bob/)
2. Project rules (from project/.bob/) - these can override global rules
3. Legacy files (.bobrules, .clinerules - for backward compatibility)

Within each level, mode-specific rules are loaded before general rules.

Rules file guidelines

- **File location:** Preferred method uses directories within `.bob/` (`.bob/rules/` and `.bob/rules-{modeSlug}/`). Fallback method uses single files (`.bobrules` and `.bobrules-{modeSlug}`) in the workspace root.
- **Recursive reading:** Rules directories are read recursively, including all files in subdirectories
- **File filtering:** System automatically excludes cache and temporary files (.DS_Store, *.bak, *.cache, *.log, *.tmp, Thumbs.db, etc.)
- **Empty files:** Empty or missing rule files are silently skipped
- **Source headers:** Directory-based rules are included without headers, while file-based rules include `# Rules from {filename}:` headers
- **Rule interaction:** Mode-specific rules complement global rules rather than replacing them
- **Symbolic links:** Fully supported for both files and directories, with a maximum resolution depth of 5 to prevent infinite loops

AGENTS.md integration

Bob can load rules from an AGENTS.md (or AGENT.md as fallback) file in your workspace root:

- **Purpose:** Defines agent-specific rules and guidelines for AI behavior
- **Location:** Must be in the workspace root directory
- **Loading:** Automatically loaded by default. To disable AGENTS.md loading, set `"bob-cline.useAgentRules": false` in your Bob settings
- **Priority:** Loaded after mode-specific rules but before general workspace rules
- **Header:** Added to system prompt with header `# Agent Rules Standard (AGENTS.md):`
- **Symbolic links:** Works with symbolic links to AGENTS.md files in other locations

Use AGENTS.md files to maintain standardized AI agent behavior rules with your team. AGENTS.md file can be version-controlled alongside your project code.

Custom instruction examples

- "Always use spaces for indentation, with a width of 4 spaces"
- "Use camelCase for variable names"
- "Write unit tests for all new functions"

- "Explain your reasoning before providing code"
- "Focus on code readability and maintainability"
- "Prioritize using the most common library in the community"
- "When adding new features to websites, ensure they are responsive and accessible"

Team standardization strategies

For team environments, consider these approaches:

- **Project standards:** Use workspace `.bob/rules/` directories under version control to standardize Bob's behavior for specific projects. This ensures consistent code style and development workflows across team members.
- **Organization standards:** Use global rules (`~/.bob/rules/`) to establish organization-wide coding standards that apply to all projects. Team members can set up identical global rules for consistency across all work.
- **Hybrid approach:** Combine global rules for organization standards with project-specific workspace rules for project-specific requirements. Workspace rules can override global rules when needed.

The directory-based approach offers better organization than single `.bobrules` files and enables both global and project-level customization.

Advanced customization with modes

For more advanced customization, combine custom instructions with Custom modes to create specialized environments with specific tool access, file restrictions, and tailored instructions.

How tools work

Bob uses tools to interact with your code and environment. These specialized helpers perform specific actions like reading files, making edits, running commands, or searching your codebase. Tools automate common development tasks without requiring manual execution.

Tool workflow

When you describe what you want to accomplish in natural language, Bob will:

1. Select the appropriate tool based on your request
2. Present the tool with its parameters for your review
3. Execute the approved tool and show you the results
4. Continue this process until your task is complete

Tool categories

Category	Purpose	Tool Names
Read	Access file content and code structure	`read_file`, `search_files`, `list_files`, `list_code_definition_names`

Category	Purpose	Tool Names
Write	Create or modify files and code	`write_to_file`, `apply_diff`, `insert_content`, `search_and_replace`
Browser	Interact with web content	`browser_action`
Retry	Retry failed API requests	`retry`
MCP	Use MCP tools	`mcp`
Mode	Switch to a different mode	`mode`
Subtasks	Create and complete subtasks	`subtask`
Execute	Run commands and perform system operations	`execute_command`
Question	Ask a question about the task	`question`

Context mentions

Context mentions let you reference specific elements of your project directly in your conversations with Bob. By using the `@` symbol, you can point Bob to files, folders, problems, and other project components, enabling more accurate and efficient assistance.

Types of mentions

With context mentions, you can reference various elements of your project:

Mention Type	Format	Description	Example Usage
File	`@/path/to/file.ts`	Includes file contents in request context	"Explain the function in @/src/utils.ts"
Folder	`@/path/to/folder`	Includes contents of all files directly in the folder (non-recursive)	"Analyze the code in @/src/components"
Problems	`@problems`	Includes VS Code Problems panel diagnostics	"@problems Fix all errors in my code"
Terminal	`@terminal`	Includes recent terminal command and output	"Fix the errors shown in @terminal"
Git Commit	`@a1b2c3d`	References specific commit by hash	"What changed in commit @a1b2c3d?"
Git Changes	`@git-changes`	Shows uncommitted changes	"Suggest a message for @git-changes"
URL	`@https://example.com`	Imports website content	"Summarize @https://docusaurus.io/"

File mentions

File mentions are the most commonly used type, letting you include the contents of specific files in your conversation with Bob.

Capability	Details
Format	`@/path/to/file.ts` (always start with `/` from workspace root)
Provides	Complete file contents with line numbers
Works with	Text files, PDFs, and DOCX files (with text extraction)

Capability	Details
Works in	Initial requests, feedback responses, and follow-up messages
Limitations	Very large files may be truncated; binary files not supported

> **Bob's tip:** For large files, you can mention specific sections by using the file's path followed by line numbers, like `@/src/app.js:10-20` to include only lines 10-20.

Folder mentions

With folder mentions, you can reference multiple files at once, providing broader context from a directory.

Capability	Details
Format	`@/path/to/folder` (no trailing slash)
Provides	Complete contents of all files within the directory
Includes	Contents of non-binary text files directly within the folder (not recursive)
Best for	Providing context from multiple files in a directory
Tip	Be mindful of context window limits when mentioning large directories

> **Note:** Folder mentions are non-recursive, meaning they only include files directly in the specified folder, not in subfolders.

Problems mention

The problems mention imports diagnostics from VS Code's Problems panel, helping you resolve multiple issues at once.

Capability	Details
Format	`@problems`
Provides	All errors and warnings from VS Code's problems panel
Includes	File paths, line numbers, and diagnostic messages
Groups	Problems organized by file for better clarity
Best for	Fixing errors without manual copying

Example usage: "@problems Can you help me fix all these issues?"

Terminal mention

Terminal mentions capture recent command output, helping you debug build errors or analyze command results.

Capability	Details
Format	`@terminal`
Captures	Last command and its complete output
Preserves	Terminal state (doesn't clear the terminal)
Limitation	Limited to visible terminal buffer content
Best for	Debugging build errors or analyzing command output

> **Bob's tip:** When troubleshooting a build failure, try "What's wrong with @terminal?" to get immediate insights into the error without manually copying the output.

Git mentions

Git mentions help you work with version control by referencing commits and changes.

Type	Format	Provides	Limitations
Commit	`@a1b2c3d`	Commit message, author, date, and complete diff	Only works in Git repositories
Working Changes	`@git-changes`	`git status` output and diff of uncommitted changes	Only works in Git repositories

Example usage: "What's the purpose of @a1b2c3d?" or "Suggest a commit message for @git-changes"

URL mentions

URL mentions let you reference external web content in your conversations with Bob.

Capability	Details
Format	`@https://example.com`
Processing	Uses headless browser to fetch content
Cleaning	Removes scripts, styles, and navigation elements
Output	Converts content to Markdown for readability
Limitation	Complex pages may not convert perfectly

> **Bob's tip:** URL mentions are useful for referencing documentation. Try "Can you explain @https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise" to get help understanding a concept with the official docs as context.

How to use mentions

Using context mentions is straightforward:

1. Type `@` in the chat input to trigger the suggestions dropdown
2. Continue typing to filter suggestions or use arrow keys to navigate
3. Select with Enter key or mouse click
4. Combine multiple mentions in a request: "Fix @problems in @/src/component.ts"

The dropdown automatically suggests:

- Recently opened files
- Visible folders
- Recent git commits
- Special keywords (`problems`, `terminal`, `git-changes`)
- All currently open files (regardless of ignore settings or directory filters)

The dropdown automatically filters out common directories like `node_modules`, `.git`, `dist`, and `out` to reduce noise, even though you can include their content if manually typed.

> **Pro tip:** You can combine multiple mentions in a single request. Try "Compare @/src/v1/api.js with @/src/v2/api.js and explain the differences."

Important behaviors

Ignore file interactions

Context mentions have specific behaviors when interacting with ignored files:

Behavior	Description
`.bobignore` bypass	File and folder `@mentions` bypass `.bobignore` checks when fetching content for context. Content from ignored files will be included if directly mentioned.
`.gitignore` bypass	Similarly, file and folder `@mentions` do not respect `.gitignore` rules when fetching content.
Git command respect	Git-related mentions (`@git-changes`, `@commit-hash`) do respect `.gitignore` since they rely on Git commands.

This means you can reference specific files that would normally be ignored by Bob or Git, which is particularly useful when you need to discuss generated files or other typically ignored content.

Putting it all together

Context mentions enhance your interactions with Bob by providing precise references to your project's elements. Instead of copying and pasting code or describing file locations, you can directly reference what you're discussing.

Effective combinations of different mention types:

- "Fix @problems in @/src/components and explain what was wrong"
- "Review @git-changes and suggest improvements"
- "Compare @/v1/api.js with @https://api-docs.example.com and check for inconsistencies"

By using context mentions effectively, you can communicate more precisely with Bob and receive more targeted assistance.

Using .bobignore to Control File Access

The **`.bobignore`** file is a key feature for managing Bob's interaction with your project files. It allows you to specify files and directories that Bob should not access or modify, similar to how **`.gitignore`** works for Git.

What is `.bobignore`?

- **Purpose**: To protect sensitive information, prevent accidental changes to build artifacts or large assets, and generally define Bob's operational scope within your workspace.
- **How to Use**: Create a file named **`.bobignore`** in the Bob directory of your VS Code workspace. List patterns in this file to tell Bob which files and directories to ignore.

Bob actively monitors the **`.bobignore`** file. Any changes you make are reloaded automatically, ensuring Bob always uses the most current rules. The **`.bobignore`** file itself is always implicitly ignored, so Bob cannot change its own access rules.

Pattern Syntax

The syntax for **`.bobignore`** is identical to **`.gitignore`**. Here are common examples:

- **`node_modules/`**: Ignores the entire **`node_modules`** directory.
- **`*.log`**: Ignores all files ending in **`.log`**.
- **`config/secrets.json`**: Ignores a specific file.

- `!important.log`: An exception; Bob will *not* ignore this specific file, even if a broader pattern like `*.log` exists.
- `build/`: Ignores the `build` directory.
- `docs/**/*.md`: Ignores all Markdown files in the `docs` directory and its subdirectories.

For a comprehensive guide on syntax, refer to the official Git documentation on .gitignore.

How Bob Tools Interact with `.bobignore`

`.bobignore` rules are enforced across various Bob tools:

Strict Enforcement (Reads & Writes)

These tools directly check `.bobignore` before any file operation. If a file is ignored, the operation is blocked:

- `read_file`: Will not read ignored files.
- `write_to_file`: Will not write to or create new ignored files.
- `apply_diff`: Will not apply diffs to ignored files.
- `list_code_definition_names`: Will not parse ignored files for code symbols.

File Editing Tools (Potential Write Bypass)

The `insert_content` and `search_and_replace` tools use an internal component for managing changes. **Important**: Currently, the final write operation performed by these tools might bypass `.bobignore` rules. While initial read attempts might be blocked, the save action itself does not have an explicit check.

File Discovery and Listing

- `list_files` **Tool**: When Bob lists files, ignored files are typically omitted or marked with a symbol (see "User Experience" below).
- **Environment Details**: Information about your workspace (like open tabs and project structure) provided to Bob is filtered to exclude or mark ignored items.

Command Execution

- `execute_command` **Tool**: This tool checks if a command (from a predefined list like `cat` or `grep`) targets an ignored file. If so, execution is blocked.

Key Limitations and Scope

- **Workspace-Centric**: `.bobignore` rules apply **only to files and directories within the current VS Code workspace Bobt**. Files outside this scope are not affected.
- `execute_command` **Specificity**: Protection for `execute_command` is limited to a predefined list of file-reading commands. Custom scripts or uncommon utilities might not be caught.
- **Write Operations via** `insert_content` **&** `search_and_replace`: As noted, these tools might be able to write to ignored files due to current limitations in their save mechanism.
- **Not a Full Sandbox**: `.bobignore` is a powerful tool for controlling Bob's file access via its tools, but it does not create a system-level sandbox.

User Experience and Notifications

- **Visual Cue ( )**: In file listings, files ignored by `.bobignore` may be marked with a lock symbol ( ), depending on the `showbobignoredFiles` setting (defaults to `true`).

- **Error Messages**: If a tool operation is blocked, Bob receives an error: `"Access to [file_path] is blocked by the .bobignore file settings. You must try to continue in the task without using this file, or ask the user to update the .bobignore file."`
- **Chat Notifications**: You will typically see a notification in the Bob chat interface when an action is blocked due to `.bobignore`.

This guide helps you understand the `.bobignore` feature, its capabilities, and its current limitations, so you can effectively manage Bob's interaction with your codebase.

Auto-approving actions

> ⚠ **SECURITY WARNING:** Auto-approve settings bypass confirmation prompts, giving Bob direct access to your system. This can result in **data loss, file corruption, or worse**. Command line access is particularly dangerous, as it can potentially execute harmful operations that could damage your system or compromise security. Only enable auto-approval for actions you fully trust.

Auto-approve settings speed up your workflow by eliminating repetitive confirmation prompts, but they significantly increase security risks.

Quick start guide

1. Click the Auto-Approve toolbar above the chat input
2. Select which actions Bob can perform without asking permission

Available permissions

Permission	What it does	Risk level
Read files and directories	Allows Bob to access files without asking	Medium
Edit files	Allows Bob to modify files without asking	**High**
Execute approved commands	Runs whitelisted terminal commands automatically	**High**
Use the browser	Allows headless browser interaction	Medium
Use MCP servers	Allows Bob to use configured MCP services	Medium-High
Switch modes	Changes between Bob modes automatically	Low
Create & complete subtasks	Manages subtasks without confirmation	Low
Retry failed requests	Automatically retries failed API requests	Low

When to use auto-approving actions

Auto-approving actions are most beneficial in these scenarios:

- **Repetitive development tasks** where you trust Bob's actions, such as generating boilerplate code
- **Batch operations** where you need to process multiple files without interruption
- **Exploratory coding sessions** where you want to maintain flow without constant prompts
- **Local development environments** where security risks are contained

Best practices for security

Follow these guidelines to use auto-approving actions safely:

- **Start restrictive** - Begin with minimal permissions and add more only as needed
- **Use project-specific settings** - Customize auto-approve settings for each project based on risk tolerance
- **Disable when not needed** - Turn off auto-approve when working with sensitive code or production systems
- **Review changes regularly** - Periodically check what actions Bob has taken, especially file modifications
- **Never auto-approve in production** - Restrict auto-approve settings to development environments only

Using Bob tips

Get refactoring suggestions as you code or view files with Bob tips. Bob uses different techniques to identify complex functions, classes, and patterns within your code and then offers tips for refactoring. Bob tips helps improve your code by providing suggestions to enhance quality and efficiency, while reducing maintenance requirements.

You can get Bob tips in one of the following formats:

- Direct fix: Use Bob to refactor your code directly in your file. Use the direct fix option to quickly apply Bob's suggestion.
- Fix in chat: Use the chat interface to collaborate with Bob to refactor your code. Use the fix in chat option to create a plan with Bob and implement your solution using agentic flow.

To disable Bob tips:

1. Open the Bob - Settings panel by clicking the **Bob - Settings** button in the bottom right corner.
2. Click the toggle to turn the feature on or off.

Code Actions

Code Actions are a powerful feature of VS Code that provide quick fixes, refactorings, and other code-related suggestions directly within the editor. Bob integrates with this system to offer AI-powered assistance for common coding tasks.

What are Code Actions?

Code Actions appear as a lightbulb icon ( ) in the editor gutter (the area to the left of the line numbers). They can also be accessed via the right-click context menu, or via keyboard shortcut. They are triggered when:

- You select a range of code.
- Your cursor is on a line with a problem (error, warning, or hint).
- You invoke them via command.

Clicking the lightbulb, right-clicking and selecting "Bob", or using the keyboard shortcut (`Ctrl+.` or `Cmd+.` on macOS, by default), displays a menu of available actions.

Bob's Code Actions

Bob provides the following Code Actions:

- **Add to Context:** Quickly adds the selected code to your chat with Bob, including the filename and line numbers so Bob knows exactly where the code is from. It's listed first in the menu for easy access.
- **Explain Code:** Asks Bob to explain the selected code.
- **Improve Code:** Asks Bob to suggest improvements to the selected code.

Add to Context Deep Dive

The **Add to Context** action is listed first in the Code Actions menu so you can quickly add code snippets to your conversation. When you use it, Bob includes the filename and line numbers along with the code.

This helps Bob understand the exact context of your code within the project, allowing it to provide more relevant and accurate assistance.

Example Chat Input:

```
Can you explain this function?
@myFile.js:15:25
```

(Where `@myFile.js:15:25` represents the code added via "Add to Context")

Using Code Actions

There are three main ways to use Bob's Code Actions:

1. From the Lightbulb ( )

1. **Select Code:** Select the code you want to work with. You can select a single line, multiple lines, or an entire block of code.
2. **Look for the Lightbulb:** A lightbulb icon will appear in the gutter next to the selected code (or the line with the error/warning).
3. **Click the Lightbulb:** Click the lightbulb icon to open the Code Actions menu.
4. **Choose an Action:** Select the desired Bob action from the menu.
5. **Review and Approve:** Bob will propose a solution in the chat panel. Review the proposed changes and approve or reject them.

2. From the Right-Click Context Menu

1. **Select Code:** Select the code you want to work with.
2. **Right-Click:** Right-click on the selected code to open the context menu.
3. **Choose "Bob":** Select the "Bob" option from the context menu. A submenu will appear with the available Bob actions.
4. **Choose an Action:** Select the desired action from the submenu.
5. **Review and Approve:** Bob will propose a solution in the chat panel. Review the proposed changes and approve or reject them.

3. From the Command Palette

1. **Select Code:** Select the code you want to work with.
2. **Open the Command Palette:** Press `Ctrl+Shift+P` (Windows/Linux) or `Cmd+Shift+P` (macOS).
3. **Type a Command:** Type "Bob" to filter the commands, then choose the relevant code action (e.g., "Bob: Explain Code"). The action will apply in the most logical context (usually the current active chat

task, if one exists).
4. **Review and Approve:** Bob will propose a solution in the chat panel. Review the proposed changes and approve or reject them.

Customizing Code Action Prompts

You can customize the prompts used for each Code Action by modifying the "Support Prompts" in the **Prompts** tab. This allows you to fine-tune the instructions given to the AI model and tailor the responses to your specific needs.

1. **Open the Prompts Tab:** Click the icon in the Bob top menu bar.
2. **Find "Support Prompts":** You will see the support prompts, including "Enhance Prompt", "Explain Code", and "Improve Code".
3. **Edit the Prompts:** Modify the text in the text area for the prompt you want to customize. You can use placeholders like `${filePath}` and `${selectedText}` to include information about the current file and selection.
4. **Click "Done":** Save your changes.

By using Bob's Code Actions, you can quickly get AI-powered assistance directly within your coding workflow. This can save you time and help you write better code.

Code reviews

Bob can review your code directly from your IDE, catching potential issues before you commit your work.

Why use this feature?

- **Catch errors early**: Identify issues before they make it into your committed work.
- **Save time**: Address potential comments that reviewers might leave, reducing the time needed for PR approval.
- **Improve code quality**: Get suggestions for better coding practices and maintainability.

How it works

When you have uncommitted changes in your workspace, you can ask Bob to review them. Bob analyzes your changes and flags potential issues in the Bob Findings panel.

You can click on any finding to get more details and work with Bob to fix the issue. Findings can be referenced in two ways:

- From the Bob Findings panel
- By clicking on @issues in the context dropdown menu in the chat interface

Getting started

Prerequisites

- You must have uncommitted changes in your workspace

Basic usage

You can initiate a code review in two ways:

- Use the command palette:
    1. Press `Cmd+Shift+P` on Mac or `Ctrl+Shift+P` on Windows/Linux.
    2. Search for "Bob: Review Current Changes".
- Use the following slash command in the chat interface:

  `/review`

Bob will analyze your changes and display findings in the Bob Findings panel.

Generating commit messages

Bob can automatically generate meaningful commit messages based on your staged changes, saving you time and ensuring consistency in your commit history.

Why use this feature?

- **Save time**: Generate well-formatted commit messages with a single click
- **Maintain consistency**: Follow conventional commit standards automatically
- **Match project style**: Bob analyzes your branch name and commit history to match your project's conventions
- **Iterate easily**: Generate alternative suggestions if you don't like the first one

How it works

Bob examines your staged changes to understand what modifications you've made to your codebase. It then analyzes your branch name and recent commit history to determine the appropriate commit message format that matches your project's style.

Getting started

Prerequisites

Your changes must be staged in the Source Control panel.

Basic usage

To automatically generate a commit message:

1. Click the sparkle (✨) icon next to the commit message box.
2. Review your commit message made by Bob.
    - Click the sparkle (✨) icon again for alternative suggestions. Apply edits to your message, if needed.
3. Commit your changes.

Tips and best practices

- For the best results, make atomic commits (changes that address a single concern).
- If you're working on a specific issue, include the issue number in your branch name.
- Edit the generated message if needed to add more specific details.
- Use the regenerate option if the first suggestion doesn't capture your changes accurately.

Generating pull requests

Bob can generate pull requests (PRs) and PR descriptions directly from your IDE, streamlining your development workflow and saving time.

Why use this feature?

- **Save time**: Generate detailed PR descriptions automatically based on your changes.
- **Maintain consistency**: Ensure all PRs follow a consistent format.
- **Improve collaboration**: Provide clear context for reviewers with well-structured descriptions.
- **Streamline workflow**: Create PRs without leaving your IDE.

How it works

Bob analyzes your branch changes, commit history, and branch name to understand the purpose and scope of your work. It then generates a meaningful PR title and detailed description that summarizes your changes, making it easier for reviewers to understand the context and purpose of your PR. After generation, you can review and edit the PR description before submission.

The PR generation process is interactive, allowing you to select the target branch and remote repository before creating the PR.

Getting started

Prerequisites

- Your changes must be committed to a branch.
- You must have push access to the remote repository.

Basic usage

You can choose from 3 options to initiate a PR flow:

- Click the PR icon in the Source Control panel.
- Use the command palette (Cmd+Shift+P on Mac, Ctrl+Shift+P on Windows/Linux) and search for "Bob: Generate PR".
- Type `/create-pr` in the Bob chat interface.

Bob will then guide you through the following steps:

1. Select the base branch you want to merge into. Bob will suggest the three most likely branches.
2. Select the remote repository, if you have multiple remotes.
3. Review the generated PR description and title.
4. Create the PR.
5. Get the newly created PR link in your chat.

Using templates for PR descriptions

Bob automatically detects and uses your project's existing PR templates when generating pull request descriptions. This ensures your PRs are consistent with your team's established standards. When Bob detects multiple PR templates in your project, you can select which template to apply to your pull request.

Template detection

Bob searches for PR templates in the following locations relative to your project root:

1. `${cwd}/pull_request_template.md`
2. `${cwd}/docs/pull_request_template.md`
3. `${cwd}/.github/pull_request_template.md`
4. `${cwd}/.github/PULL_REQUEST_TEMPLATE/pull_request_template.md`
5. `${cwd}/PULL_REQUEST_TEMPLATE/pull_request_template.md`
6. `${cwd}/docs/PULL_REQUEST_TEMPLATE/pull_request_template.md`

If Bob finds multiple templates, Bob will ask the user to choose which template to use for the PR description.

Default template

If no template is found in your project, Bob uses a built-in template that includes:

- A descriptive title derived from your branch name
- A summary of changes
- Key implementation details
- Testing information
- Any relevant links or references

Editing the generated description

After generating the PR description, you can choose to edit it before submission:

1. When prompted, select whether you want to edit the description.
2. If you choose to edit, Bob opens the description in a temporary markdown file.
3. Make your desired changes to the description.
4. Click the "Done" button when you've completed your edits.

If you choose not to edit, Bob uses the generated description as-is.

Customizing templates

For the best results with Bob's PR generation:

- Use clear section headings in your templates (e.g., `## Summary`, `## Changes`)
- Include placeholder text that provides context (e.g., `<!-- Describe the purpose of this PR -->`)
- Consider adding sections for testing instructions and related issues

Tips and best practices

- Use descriptive branch names to help Bob generate more accurate PR descriptions.
- Review and edit the generated description before creating the PR.

Enhance prompt

With the Enhance Prompt feature, you can improve the quality and effectiveness of your prompts before sending them to the AI model. By clicking the icon in the chat input, you automatically refine your initial request, making it clearer, more specific, and more likely to produce the results you want.

Why use this feature?

- **Improved clarity:** Your prompts become more understandable for the AI model
- **Added context:** The enhancement process uses relevant context such as the current file path or selected code
- **Better instructions:** Your enhanced prompt contains specific guidance for the AI, such as formatting preferences or detail level.
- **Reduced ambiguity:** The enhancement process eliminates unclear language and ensures Bob understands your intent
- **Consistency:** Your prompts follow a standardized format that works well with the AI

How it works

1. **Type your initial prompt** in the Bob chat input box as you normally would
2. **Click the icon** located in the bottom right of the chat input box
3. **Review the enhanced prompt** to make sure it accurately reflects your intent (you can further refine it if needed)
4. **Send the enhanced prompt** by pressing Enter or clicking the Send icon ( )

   Bob's tip: The enhancement process works best with prompts that have a clear goal but need additional structure or context.

Example transformations

Basic example

Original prompt:

```
make a function to calculate tax
```

Enhanced prompt:

```
Create a JavaScript function that calculates sales tax. The function should:
1. Take two parameters: price (number) and taxRate (decimal)
2. Return the calculated tax amount
3. Include proper error handling for invalid inputs
4. Include comments explaining the calculation logic

Please provide the complete function implementation with an example of how to use
it.
```

Complex example

Original prompt:

```
analyze this code for bugs
```

Enhanced prompt:

```
Please analyze the currently selected code for potential bugs and issues. Focus
on:

1. Logic errors and edge cases
2. Performance concerns
3. Security vulnerabilities
4. Error handling improvements
5. Code style and best practices

For each issue found, please:
- Explain the problem clearly
- Show the problematic code
- Provide a corrected version
- Explain why the correction resolves the issue

Present your analysis in a structured format with clear headings for each category
of issues.
```

Customizing the enhancement process

You can modify the prompt template used for enhancement to tailor it to your specific needs:

1. **Open the Prompts tab:** Click the icon in the Bob top menu bar
2. **Select the "ENHANCE" tab:** You'll see the current enhancement prompt template
3. **Edit the prompt template:** Modify the text in the "Prompt" field

The default template includes the placeholder `${userInput}`, which will be replaced with your original prompt. You can customize the instructions around this placeholder to guide how the enhancement should work.

Customization examples

For more technical enhancements:

```
Transform the following user request into a detailed technical prompt that will
produce optimal results from an AI assistant. Add specific technical details,
parameters, and expected output formats where appropriate. Original request:
${userInput}
```

For more creative enhancements:

```
Enhance the following prompt to encourage creative and innovative responses while
maintaining technical accuracy. Add context about exploring multiple approaches
and considering non-obvious solutions. Original request: ${userInput}
```

API configuration

The API configuration used for Enhance Prompt is, by default, the same one selected for Bob tasks, but you can change it:

1. **Open the Prompts tab:** Click the icon in the Bob top menu bar
2. **Select the "ENHANCE" tab:** Locate the "API Configuration" dropdown
3. **Select an API configuration:** Choose an existing configuration for future Enhance Prompt requests

   Note: Using a different model for enhancement than for your main Bob tasks can optimize for speed (with a faster model for enhancements) or cost (with a less expensive model).

Limitations and best practices

- **Experimental feature:** Prompt enhancement is still experimental. The quality of enhanced prompts may vary depending on your request's complexity and the model's capabilities.
- **Review carefully:** Always review the enhanced prompt before sending it. The enhancement process might make changes that don't align with your intentions.
- **Iterative refinement:** You can apply Enhance Prompt multiple times to iteratively refine your prompt until it's exactly what you want.
- **Start with clear intentions:** While enhancement improves your prompts, starting with a clear idea of what you want will produce the best results.
- **Context matters:** Enhancement works best when Bob has context about your current task. If you're starting a new conversation, provide some initial context before using Enhance Prompt.

By using Enhance Prompt, you can get more accurate and helpful responses from Bob while saving time on writing detailed prompts from scratch.

Keyboard Shortcuts

The Bob interface supports keyboard shortcuts to streamline your workflow and reduce dependence on mouse interactions.

Available Keyboard Commands

Bob offers keyboard commands to enhance your workflow. This page focuses on the **`Bob.acceptInput`** command, but here's a quick reference to all keyboard commands:

Command	Description	Default Shortcut
`Bob.acceptInput`	Submit text or accept the primary suggestion	None (configurable)
`Bob.focus`	Focus the Bob input box	None (configurable)

Key Benefits of Keyboard Commands

- **Keyboard-Driven Interface**: Submit text or select the primary suggestion button without mouse interaction
- **Improved Accessibility**: Essential for users with mobility limitations or those who experience discomfort with mouse usage
- **Vim/Neovim Compatibility**: Supports seamless transitions for developers coming from keyboard-centric environments
- **Workflow Efficiency**: Reduces context switching between keyboard and mouse during development tasks

Bob.acceptInput Command

The **`Bob.acceptInput`** command lets you submit text or accept suggestions with keyboard shortcuts instead of clicking buttons or pressing Enter in the input area.

What It Does

The **`Bob.acceptInput`** command is a general-purpose input submission command. When triggered, it:

- Submits your current text or image input when in the text input area (equivalent to pressing Enter)
- Clicks the primary (first) button when action buttons are visible (such as confirm/cancel buttons or any other action buttons)

Detailed Setup Guide

Method 1: Using the VS Code UI

1. Open the Command Palette (**`Ctrl+Shift+P`** or **`Cmd+Shift+P`** on Mac)
2. Type "Preferences: Open Keyboard Shortcuts"
3. In the search box, type "Bob.acceptInput"
4. Locate "Bob: Accept Input/Suggestion" in the results
5. Click the + icon to the left of the command
6. Press your desired key combination (e.g., **`Ctrl+Enter`** or **`Alt+Enter`**)
7. Press Enter to confirm

Method 2: Editing keybindings.json directly

1. Open the Command Palette (**`Ctrl+Shift+P`** or **`Cmd+Shift+P`** on Mac)
2. Type "Preferences: Open Keyboard Shortcuts (JSON)"
3. Add the following entry to the JSON array:

```
{
  "key": "ctrl+enter",  // or your preferred key combination
  "command": "Bob.acceptInput",
  "when": "BobViewFocused"  // This is a context condition that ensures the
command only works when Bob is focused
}
```

You can also use a more specific condition:

```
{
  "key": "ctrl+enter",
  "command": "Bob.acceptInput",
  "when": "webviewViewFocus && webviewViewId == 'Bob-cline.SidebarProvider'"
}
```

Recommended Key Combinations

Choose a key combination that doesn't conflict with existing VS Code shortcuts:

- **`Alt+Enter`** - Easy to press while typing
- **`Ctrl+Space`** - Familiar for those who use autocomplete
- **`Ctrl+Enter`** - Intuitive for command execution
- **`Alt+A`** - Mnemonic for "Accept"

Practical Use Cases

Quick Development Workflows

- **Text Submission**: Send messages to Bob without moving your hands from the keyboard
- **Action Confirmations**: Accept operations like saving files, running commands, or applying diffs

- **Multi-Step Processes**: Move quickly through steps that require confirmation or input
- **Consecutive Tasks**: Chain multiple tasks together with minimal interruption

Keyboard-Centric Development

- **Vim/Neovim Workflows**: If you're coming from a Vim/Neovim background, maintain your keyboard-focused workflow
- **IDE Integration**: Use alongside other VS Code keyboard shortcuts for a seamless experience
- **Code Reviews**: Quickly accept suggestions when reviewing code with Bob
- **Documentation Writing**: Submit text and accept formatting suggestions when generating documentation

Accessibility Use Cases

- **Hand Mobility Limitations**: Essential for users who have difficulty using a mouse
- **Repetitive Strain Prevention**: Reduce mouse usage to prevent or manage repetitive strain injuries
- **Screen Reader Integration**: Works well with screen readers for visually impaired users
- **Voice Control Compatibility**: Can be triggered via voice commands when using voice control software

Accessibility Benefits

The **`Bob.acceptInput`** command was designed with accessibility in mind:

- **Reduced Mouse Dependence**: Complete entire workflows without reaching for the mouse
- **Reduced Physical Strain**: Helps users who experience discomfort or pain from mouse usage
- **Alternative Input Method**: Supports users with mobility impairments who rely on keyboard navigation
- **Workflow Optimization**: Particularly valuable for users coming from keyboard-centric environments like Vim/Neovim

Keyboard-Centric Workflows

Here are some complete workflow examples showing how to effectively use keyboard shortcuts with Bob:

Development Workflow Example

1. Open VS Code and navigate to your project
2. Open Bob via the sidebar
3. Type your request: "Create a REST API endpoint for user registration"
4. When Bob asks for framework preferences, use your **`Bob.acceptInput`** shortcut to select the first suggestion
5. Continue using the shortcut to accept code generation suggestions
6. When Bob offers to save the file, use the shortcut again to confirm
7. Use VS Code's built-in shortcuts to navigate through the created files

Code Review Workflow

1. Select code you want to review and use VS Code's "Copy" command
2. Ask Bob to review it: "Review this code for security issues"
3. As Bob asks clarifying questions about the code context, use your shortcut to accept suggestions
4. When Bob provides improvement recommendations, use the shortcut again to accept implementation suggestions

Troubleshooting

Issue	Solution
Shortcut doesn't work	Ensure Bob is focused (click in the Bob panel first)
Wrong suggestion selected	The command always selects the first (primary) button; use mouse if you need a different option
Conflicts with existing shortcuts	Try a different key combination in VS Code keyboard settings
No visual feedback when used	This is normal - the command silently activates the function without visual confirmation
Shortcut works inconsistently	Make sure the `when` clause is properly configured in your keybindings.json (either `BobViewFocused` or the webview-specific condition)

Technical Implementation

The `Bob.acceptInput` command is implemented as follows:

- Command registered as `Bob.acceptInput` with display title "Bob: Accept Input/Suggestion" in the command palette
- When triggered, it sends an "acceptInput" message to the active Bob webview
- The webview determines the appropriate action based on the current UI state:
  - Clicks the primary action button if action buttons are visible and enabled
  - Sends the message if the text area is enabled and contains text/images
- No default key binding - users assign their preferred shortcut

Limitations

- Works only when the Bob interface is active
- Has no effect if no inputs or suggestions are currently available
- Prioritizes the primary (first) button when multiple options are shown

Literate coding

Literate coding lets you write code with AI assistance directly inside your editor. Instead of switching to a chat window or writing long prompts, you type instructions in plain language right where the code should go. Bob then generates the implementation for you in context and shows a diff of the changes.

Why use literate coding?

- **Stay in your workflow**: Write code without context switching to chat interfaces
- **Maintain precise control**: Specify exactly where and how AI should modify your code
- **Express intent naturally**: Describe what you want in plain language or pseudocode
- **See changes before committing**: Review diffs to ensure the generated code meets your needs

How it works

When you activate literate coding mode, you can write natural language instructions, pseudocode, annotations, or partial code snippets directly in your source file. These instructions appear highlighted in blue to distinguish them from regular code.

When you run the Generate command, Bob:

1. Analyzes the entire file including your literate coding blocks
2. Understands the surrounding context and your intent
3. Replaces your instructions with real, executable code
4. Shows you an inline diff so you can review the changes

This approach combines the expressiveness of natural language with the precision of direct code editing.

Getting started

Basic usage

1. Toggle literate coding mode:
    - Press **Cmd+M** (Mac) or **Ctrl+M** (Windows/Linux)
    - Or click the magic wand icon in the editor toolbar
2. Write your instructions:
    - Type pseudocode, natural language instructions, or annotations
    - Your text appears in blue, marking it as literate coding content
3. Generate code:
    - Press **Cmd+Enter** (recommended) or click the **Generate** button
    - Bob converts your instructions into real code and shows an inline diff
4. Review and apply:
    - **Accept** changes: `Cmd+Enter`
    - **Reject** changes: `Cmd+Shift+Backspace`
    - Changes are written to your file before acceptance, allowing you to test before deciding

Tips and best practices

- Be specific about your requirements in your instructions
- Include relevant context like variable types or function parameters
- For complex logic, break down your instructions into smaller, focused blocks
- Use literate coding for both creating new code and modifying existing code

Current limitations

- **Single-file scope**: Currently works on one file at a time (multi-file support planned)
- **Preview feature**: Expect ongoing improvements to styling, diffing, and AI behavior

Related features

- Code actions
- Using modes

Model Context Protocol (MCP)

The Model Context Protocol (MCP) extends Bob's capabilities by connecting to external tools and services. MCP servers provide additional functionality that helps Bob accomplish tasks beyond its built-in capabilities, such as accessing databases, interacting with custom APIs, and leveraging specialized tools.

Why use MCP?

- **Extend Bob's capabilities** with custom tools and integrations
- **Access external services** like databases, APIs, and specialized functionality
- **Create reusable tools** that follow a standardized protocol
- **Share tools** across teams and projects

MCP documentation sections

Using MCP in Bob

Comprehensive guide to configuring, enabling, and managing MCP servers with Bob. Includes:

- Setting up global and project-level configurations
- Managing server settings and tool permissions
- Creating custom MCP servers
- Troubleshooting common issues

What is MCP?

Clear explanation of the Model Context Protocol, including:

- Client-server architecture
- How MCP enables AI systems to interact with external tools
- Common questions and answers about MCP
- How MCP works in Bob

STDIO & SSE Transports

Detailed comparison of transport mechanisms for MCP servers:

- Local (STDIO) transport for running servers on your machine
- Remote (SSE) transport for accessing servers over a network
- Streamable HTTP transport (the modern standard)
- Deployment considerations for each approach

Getting started with MCP

If you're new to MCP, start with What is MCP? to understand the core concepts, then move to Using MCP in Bob for practical implementation steps.

What is MCP?

MCP (Model Context Protocol) is a standardized communication protocol for LLM systems to interact with external tools and services. It functions as a universal adapter between AI assistants and various data sources or applications.

How It Works

MCP uses a client-server architecture:

1. The AI assistant (client) connects to MCP servers
2. Each server provides specific capabilities (file access, database queries, API integrations)
3. The AI uses these capabilities through a standardized interface
4. Communication occurs via JSON-RPC 2.0 messages

Think of MCP as similar to a USB-C port in the sense that any compatible LLM can connect to any MCP server to access its functionality. This standardization eliminates the need to build custom integrations for each tool and service.

For example, an AI using MCP can perform tasks like "search our company database and generate a report" without requiring specialized code for each database system.

Common Questions

- **Is MCP a cloud service?** MCP servers can run locally on your computer or remotely as cloud services, depending on the use case and security requirements.
- **Does MCP replace other integration methods?** No. MCP complements existing tools like API plugins and retrieval-augmented generation. It provides a standardized protocol for tool interaction but doesn't replace specialized integration approaches.
- **How is security handled?** Users control which MCP servers they connect to and what permissions those servers have. As with any tool that accesses data or services, use trusted sources and configure appropriate access controls.

MCP in Bob

Bob implements the Model Context Protocol to:

- Connect to both local and remote MCP servers
- Provide a consistent interface for accessing tools
- Extend functionality without core modifications
- Enable specialized capabilities on demand

MCP provides a standardized way for AI systems to interact with external tools and services, making complex integrations more accessible and consistent.

Learn More About MCP

Ready to dig deeper? Check out these guides:

- MCP Overview - A quick glance at the MCP documentation structure
- Using MCP in Bob - Get started with MCP in Bob, including creating simple servers
- MCP vs API - Technical advantages compared to traditional APIs
- STDIO & SSE Transports - Local vs. hosted deployment models

MCP Server Transports: STDIO, Streamable HTTP, and SSE

Model Context Protocol (MCP) supports three primary transport mechanisms for communication between Bob and MCP servers:

- Standard Input/Output (STDIO)
- Streamable HTTP (the modern standard)
- Server-Sent Events (SSE) (for legacy use).

Each has distinct characteristics, advantages, and use cases.

STDIO Transport

STDIO transport runs locally on your machine and communicates via standard input/output streams.

How STDIO Transport Works

1. The client (Bob) spawns an MCP server as a child process
2. Communication happens through process streams: client writes to server's STDIN, server responds to STDOUT
3. Each message is delimited by a newline character
4. Messages are formatted as JSON-RPC 2.0

```
Client                      Server
  |                           |
  |---- JSON message ------>| (via STDIN)
  |                           | (processes request)
  |<---- JSON message ------| (via STDOUT)
  |                           |
```

STDIO Characteristics

- **Locality**: Runs on the same machine as Bob
- **Performance**: Very low latency and overhead (no network stack involved)
- **Simplicity**: Direct process communication without network configuration
- **Relationship**: One-to-one relationship between client and server
- **Security**: Inherently more secure as no network exposure

When to Use STDIO

STDIO transport is ideal for:

- Local integrations and tools running on the same machine
- Security-sensitive operations
- Low-latency requirements
- Single-client scenarios (one Bob instance per server)
- Command-line tools or IDE extensions

STDIO Implementation Example

```
import { Server } from '@modelcontextprotocol/sdk/server/index.js';
import { StdioServerTransport } from '@modelcontextprotocol/sdk/server/stdio.js';
```

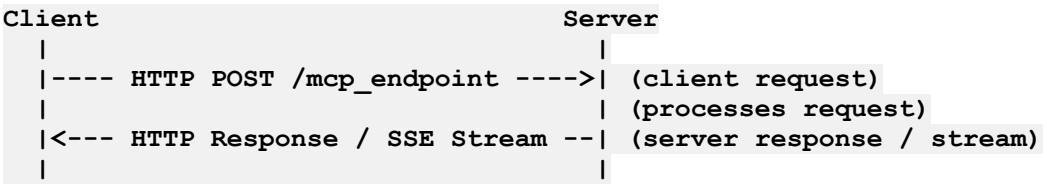
```
const server = new Server({name: 'local-server', version: '1.0.0'});
// Register tools...

// Use STDIO transport
const transport = new StdioServerTransport(server);
transport.listen();
```

Streamable HTTP Transport

Streamable HTTP transport is the modern standard for remote MCP server communication, replacing the older HTTP+SSE transport. It operates over HTTP/HTTPS and allows for more flexible server implementations.

How Streamable HTTP Transport Works

1. The server provides a single HTTP endpoint (MCP endpoint) that supports both POST and GET methods.
2. The client (Bob) sends requests to this MCP endpoint using HTTP POST.
3. The server processes the request and sends back a response.
4. Optionally, the server can use Server-Sent Events (SSE) over the same connection to stream multiple messages or notifications to the client. This allows for basic request-response interactions as well as more advanced streaming and server-initiated communication.



Streamable HTTP Characteristics

- Modern Standard: Preferred method for new remote MCP server implementations.
- Remote Access: Can be hosted on a different machine from Bob.
- Scalability: Can handle multiple client connections concurrently.
- Protocol: Works over standard HTTP/HTTPS.
- Flexibility: Supports simple request-response and advanced streaming.
- Single Endpoint: Uses a single URL path for all MCP communication.
- Authentication: Can use standard HTTP authentication mechanisms.
- Backwards Compatibility: Servers can maintain compatibility with older HTTP+SSE clients.

When to Use Streamable HTTP

Streamable HTTP transport is ideal for:

- All new remote MCP server developments.
- Servers requiring robust, scalable, and flexible communication.
- Integrations that might involve streaming data or server-sent notifications.
- Public services or centralized tools.
- Replacing legacy SSE transport implementations.

Streamable HTTP Implementation Example

Configuration in `settings.json`:

```
"mcp.servers": {
  "StreamableHTTPMCPName": {
    "type": "streamable-http",
    "url": "http://localhost:8080/mcp"
  }
}
```

For server-side implementation, refer to the MCP SDK documentation for `StreamableHTTPClientTransport`.

Backwards Compatibility with HTTP+SSE

Clients and servers can maintain backwards compatibility with the deprecated HTTP+SSE transport.

Servers wanting to support older clients should:

- Continue to host both the SSE (`/events`) and POST (`/message`) endpoints of the old transport, alongside the new “MCP endpoint” defined for the Streamable HTTP transport.

SSE Transport (Legacy)

Server-Sent Events (SSE) transport runs on a remote server and communicates over HTTP/HTTPS.

How SSE Transport Works

1. The client (Bob) connects to the server's SSE endpoint via HTTP GET request
2. This establishes a persistent connection where the server can push events to the client
3. For client-to-server communication, the client makes HTTP POST requests to a separate endpoint
4. Communication happens over two channels:
    - Event Stream (GET): Server-to-client updates
    - Message Endpoint (POST): Client-to-server requests

```
Client                                Server
  |                                     |
  |---- HTTP GET /events ------------>| (establish SSE connection)
  |<---- SSE event stream ------------| (persistent connection)
  |                                     |
  |---- HTTP POST /message ---------->| (client request)
  |<---- SSE event with response ----| (server response)
  |                                     |
```

SSE Characteristics

- **Remote Access**: Can be hosted on a different machine from Bob
- **Scalability**: Can handle multiple client connections concurrently
- **Protocol**: Works over standard HTTP (no special protocols needed)
- **Persistence**: Maintains a persistent connection for server-to-client messages
- **Authentication**: Can use standard HTTP authentication mechanisms

When to Use SSE

SSE transport is better for:

- Remote access across networks
- Multi-client scenarios
- Public services
- Centralized tools that many users need to access

- Integration with web services

SSE Implementation Example

```
import { Server } from '@modelcontextprotocol/sdk/server/index.js';
import { SSEServerTransport } from '@modelcontextprotocol/sdk/server/sse.js';
import express from 'express';

const app = express();
const server = new Server({name: 'remote-server', version: '1.0.0'});
// Register tools...

// Use SSE transport
const transport = new SSEServerTransport(server);
app.use('/mcp', transport.requestHandler());
app.listen(3000, () => {
  console.log('MCP server listening on port 3000');
});
```

Local vs. Hosted: Deployment Aspects

The choice between STDIO and SSE transports directly impacts how you'll deploy and manage your MCP servers.

STDIO: Local Deployment Model

STDIO servers run locally on the same machine as Bob, which has several important implications:

- **Installation**: The server executable must be installed on each user's machine
- **Distribution**: You need to provide installation packages for different operating systems
- **Updates**: Each instance must be updated separately
- **Resources**: Uses the local machine's CPU, memory, and disk
- **Access Control**: Relies on the local machine's filesystem permissions
- **Integration**: Easy integration with local system resources (files, processes)
- **Execution**: Starts and stops with Bob (child process lifecycle)
- **Dependencies**: Any dependencies must be installed on the user's machine

Practical Example

A local file search tool using STDIO would:

- Run on the user's machine
- Have direct access to the local filesystem
- Start when needed by Bob
- Not require network configuration
- Need to be installed alongside Bob or via a package manager

SSE: Hosted Deployment Model

SSE servers can be deployed to remote servers and accessed over the network:

- **Installation**: Installed once on a server, accessed by many users
- **Distribution**: Single deployment serves multiple clients
- **Updates**: Centralized updates affect all users immediately
- **Resources**: Uses server resources, not local machine resources
- **Access Control**: Managed through authentication and authorization systems
- **Integration**: More complex integration with user-specific resources

- **Execution**: Runs as an independent service (often continuously)
- **Dependencies**: Managed on the server, not on user machines

Practical Example

A database query tool using SSE would:

- Run on a central server
- Connect to databases with server-side credentials
- Be continuously available for multiple users
- Require proper network security configuration
- Be deployed using container or cloud technologies

Hybrid Approaches

Some scenarios benefit from a hybrid approach:

1. **STDIO with Network Access**: A local STDIO server that acts as a proxy to remote services
2. **SSE with Local Commands**: A remote SSE server that can trigger operations on the client machine through callbacks
3. **Gateway Pattern**: STDIO servers for local operations that connect to SSE servers for specialized functions

Choosing Between STDIO and SSE

Consideration	STDIO	SSE
Location	Local machine only	Local or remote
Clients	Single client	Multiple clients
Performance	Lower latency	Higher latency (network overhead)
Setup Complexity	Simpler	More complex (requires HTTP server)
Security	Inherently secure	Requires explicit security measures
Network Access	Not needed	Required
Scalability	Limited to local machine	Can distribute across network
Deployment	Per-user installation	Centralized installation
Updates	Distributed updates	Centralized updates
Resource Usage	Uses client resources	Uses server resources
Dependencies	Client-side dependencies	Server-side dependencies

Configuring Transports in Bob

For detailed information on configuring STDIO and SSE transports in Bob, including example configurations, see the Understanding Transport Types section in the Using MCP in Bob guide.

Using MCP in Bob

An MCP (Model Context Protocol) server acts as a bridge, giving Bob access to a wider range of **tools** and external services like databases, APIs, or custom scripts. It uses a standard communication method, allowing Bob to leverage these external capabilities.

For a deeper dive, check out What is MCP?.

Model Context Protocol (MCP) extends Bob's capabilities by connecting to external tools and services. This guide covers everything you need to know about using MCP with Bob.

Configuring MCP Servers

MCP server configurations can be managed at two levels:

1. **Global Configuration**: Stored in the `mcp_settings.json` file, accessible via VS Code settings (see below). These settings apply across all your workspaces unless overridden by a project-level configuration.
2. **Project-level Configuration**: Defined in a `.Bob/mcp.json` file within your project's Bobt directory. This allows you to set up project-specific servers and share configurations with your team by committing the file to version control. Bob automatically detects and loads this file if it exists.

Precedence: If a server name exists in both global and project configurations, the **project-level configuration takes precedence**.

Editing MCP Settings Files

You can edit both global and project-level MCP configuration files directly from the Bob MCP settings view:

1. Click the icon in the top navigation of the Bob pane.

   Each MCP server has its own configuration panel where you can modify settings, manage tools, and control its operation. To access these settings:

   1. Click the icon in the top navigation of the Bob pane
   2. Locate the MCP server you want to manage in the list

Auto Approve Tools

MCP tool auto-approval works on a per-tool basis and is disabled by default. To configure auto-approval:

1. First enable the global "Use MCP servers" auto-approval option in auto-approving-actions
2. In the MCP server settings, locate the specific tool you want to auto-approve
3. Check the `Always allow` checkbox next to the tool name

When enabled, Bob will automatically approve this specific tool without prompting. Note that the global "Use MCP servers" setting takes precedence - if it's disabled, no MCP tools will be auto-approved.

Finding and Installing MCP Servers

Bob does not come with any pre-installed MCP servers. You'll need to find and install them separately.

- **Community Repositories:** Check for community-maintained lists of MCP servers on GitHub
- **Ask Bob:** You can ask Bob to help you find or even create MCP servers (when "Enable MCP Server Creation" is enabled)
- **Build Your Own:** Create custom MCP servers using the SDK to extend Bob with your own tools

For full SDK documentation, visit the MCP GitHub repository.

Using MCP Tools in Your Workflow

After configuring an MCP server, Bob automatically detects its available tools and resources. Effectively leveraging these tools involves understanding the core interaction steps and, crucially, how Bob interprets the tools you provide.

Core Workflow Steps

Your interaction with MCP tools typically follows this sequence:

1. Initiate a Task

Begin by typing your request in the Bob chat interface.

2. Tool Identification by Bob

Bob analyzes your request to determine if an available MCP tool can assist. This stage is highly dependent on the quality of your MCP tool definitions.

The Critical Role of Descriptions

Bob's ability to:

- Identify the *correct* tool for the job,
- Understand how to structure the necessary parameters, and
- Avoid misinterpreting a tool's capabilities, all hinge on clear, concise, and informative descriptions for both the tools themselves and their parameters. Vague or missing information, especially for parameters, can significantly hinder Bob's ability to select or use a tool effectively.

For instance, a request like "Analyze the performance of my API" might lead Bob to consider an MCP tool designed for API endpoint testing. Whether Bob successfully identifies and utilizes this tool as intended is directly influenced by the quality of its description.

Best Practices for Defining MCP Tools

To ensure Bob can leverage your MCP tools efficiently, consider the following when defining them in your server:

- **Tool Name:** Choose a descriptive and unambiguous name that clearly indicates the tool's primary function.
- **Tool Description:** Provide a comprehensive summary of what the tool does, its purpose, and any important context or prerequisites for its use. Explain the outcome or result of using the tool.
- **Parameter Descriptions:** This is critical. For each parameter:
  - Clearly state its purpose and what kind of data it expects (e.g., "User ID for lookup," "File path to process," "Search query string").
  - Specify any formatting requirements, constraints, or an example of a valid value if applicable.
  - Indicate if the parameter is optional or required (though the MCP schema usually handles this, a note can be helpful).
- **Clarity for the AI:** Write descriptions as if you are explaining the tool to another developer (or an AI). The more context Bob has, the better it can integrate the tool into its problem-solving workflows. If a tool is intended to be used in a specific sequence or in conjunction with other tools, mentioning this can also be beneficial.

- **Augment with Custom Instructions:** Beyond the descriptions embedded in the MCP server, you can further guide Bob's usage of specific MCP tools by providing Custom Instructions. This allows you to define preferred approaches, outline complex workflows involving multiple tools, or specify when a particular MCP tool should be prioritized or avoided.

3. Tool Invocation

If Bob, guided by the tool descriptions, identifies a suitable tool, it will propose its use. You then approve this (unless auto-approval is configured for trusted tools).

Maximizing Synergy with MCP Servers

By investing effort in crafting detailed descriptions and potentially augmenting them with custom instructions, you significantly improve the synergy between Bob and your MCP servers. This unlocks their full potential for more reliable and efficient task completion.

Troubleshooting MCP Servers

Common issues and solutions:

- **Server Not Responding:** Check if the server process is running and verify network connectivity
- **Permission Errors:** Ensure proper API keys and credentials are configured in your `mcp_settings.json` (for global settings) or `.Bob/mcp.json` (for project settings).
- **Tool Not Available:** Confirm the server is properly implementing the tool and it's not disabled in settings
- **Slow Performance:** Try adjusting the network timeout value for the specific MCP server

Platform-Specific MCP Configuration Examples

Windows Configuration Example

When setting up MCP servers on Windows, you'll need to use the Windows Command Prompt (`cmd`) to execute commands. Here's an example of configuring a Puppeteer MCP server on Windows:

```
{
  "mcpServers": {
    "puppeteer": {
      "command": "cmd",
      "args": [
        "/c",
        "npx",
        "-y",
        "@modelcontextprotocol/server-puppeteer"
      ]
    }
  }
}
```

This Windows-specific configuration:

- Uses the `cmd` command to access the Windows Command Prompt
- Uses `/c` to tell cmd to execute the command and then terminate
- Uses `npx` to run the package without installing it permanently

- The `-y` flag automatically answers "yes" to any prompts during installation
- Runs the `@modelcontextprotocol/server-puppeteer` package which provides browser automation capabilities

macOS and Linux Configuration Example

When setting up MCP servers on macOS or Linux, you can use a simpler configuration since you don't need the Windows Command Prompt. Here's an example of configuring a Puppeteer MCP server on macOS or Linux:

```
{
  "mcpServers": {
    "puppeteer": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-puppeteer"
      ]
    }
  }
}
```

This configuration:

- Directly uses `npx` without needing a shell wrapper
- Uses the `-y` flag to automatically answer "yes" to any prompts during installation
- Runs the `@modelcontextprotocol/server-puppeteer` package which provides browser automation capabilities

The same approach can be used for other MCP servers on Windows, adjusting the package name as needed for different server types.

Security guidelines

Bob is a tool that provides significant capabilities for code development and system interaction. With these capabilities comes the responsibility to use Bob securely. This guide outlines essential security practices for you when working with Bob.

Security checklist

- Configure `.bobignore` to restrict file access
- Review and limit auto-approve settings
- Handle secrets securely
- Use MCP with proper authentication and encryption
- Review Bob's output before implementing

File access restrictions

You can control Bob's file access by configuring which directories and file types it can interact with. The `.bobignore` file uses the same syntax as `.gitignore` and should be one of the first security measures you implement when setting up Bob.

Setting up `.bobignore`

1. Create a `.bobignore` file in your workspace
2. Add patterns for sensitive files and directories. Include patterns for any non-approved data types.

```
# Example .bobignore patterns
.env
secrets/
*.key
config/credentials.json
```

Bob actively monitors the `.bobignore` file, and any changes are automatically applied. For detailed information, see Using .bobignore to Control File Access.

Understanding limitations

While `.bobignore` effectively controls Bob's access through its tools, it has some important limitations:

- It only applies to files within your current workspace
- Some write operations might bypass restrictions
- It doesn't create a system-level sandbox

Review the complete key limitations and scope to understand how `.bobignore` protects your files.

Auto-approve settings

With Bob, you can automatically approve various actions without confirmation prompts. While this speeds up your workflow, it significantly increases security risks.

> ⚠ **SECURITY WARNING:** Auto-approve settings bypass confirmation prompts, giving Bob direct access to your system. This can result in data loss, file corruption, or worse. Command line access is particularly dangerous, as it can potentially execute harmful operations.

High-risk auto-approve settings

Setting	Risk	Recommendation
Edit files	High	Enable only in controlled environments
Execute commands	High	Use whitelist and avoid wildcards
Use MCP servers	Medium-High	Only with trusted servers
Read files	Medium	Consider sensitive data exposure

Always review Bob's output to ensure it's accurate and that any generated code will act as intended. Never inherently trust output from any AI system.

For detailed information on each setting and its security implications, see Auto-Approving Actions.

Handling secrets securely

Never provide secrets directly to any AI system, including Bob. Even temporary inclusion of secrets in code can lead to unintended exposure.

Best practices for secrets management

- Store secrets in environment variable files
- Ensure both `.gitignore` and `.bobignore` restrict access to files that store secrets
- Use secret management tools when possible

- Follow the principle of least privilege when assigning credentials - only grant the minimum permissions necessary for each task.

Delegation of permissions

AI systems should not leverage credentials that allow them to act on your behalf without your intervention and review. When AI systems need to act on your behalf:

- Use delegation mechanisms like OAuth
- Implement time-limited tokens
- Monitor and audit all actions

Using MCP securely

Model Context Protocol (MCP) extends Bob's functionality by connecting to external tools and services. While powerful, MCP connections require careful security consideration.

MCP architecture overview

MCP uses a client-server architecture:

- Bob acts as the host containing the MCP client
- The client connects to MCP servers (local or remote)
- Servers provide additional tools and capabilities

Security requirements for MCP servers

When using MCP servers, practice the following guidelines:

- **Authentication**: Verify the identity of users accessing the server
- **Encryption**: Secure data in transit between Bob and the server
- **Access controls**: Limit what actions the server can perform
- **Auditing**: Track all actions for accountability

Remote MCP servers must adhere to the same security requirements as any traditional server infrastructure, including endpoint protection, network restrictions, and proper access controls.

For shared MCP servers, ensure proper auditability and accountability so actions can be traced.

Additional security considerations

- **Review generated content**: Always verify Bob's output before implementing it
- **Regular updates**: Keep Bob and its dependencies updated
- **Security training**: Ensure team members understand AI security risks
- **Incident response**: Have a plan for addressing potential security incidents
- **Workspace isolation**: Consider using dedicated workspaces for sensitive projects
- **Permission scoping**: Limit Bob's access to only what's needed for specific tasks

By following these security best practices, you can leverage Bob's powerful capabilities while maintaining a secure development environment.

Working with security scans

Get Bob to scan your code for potential vulnerability and security issues - especially helpful for reducing the risk of committing a secret!

No setup is required. Findings will automatically appear in your editor and the BobFindings viewer when they are found.

> **Note**: Bob reports findings for opened files. If you close a file after opening it, Bob continues to scan the file in the background.

To disable security scans:

1. Open the Bob - Settings panel by clicking the **Bob - Settings** button in the bottom right corner.
2. Click the toggle to turn the feature on or off.

Frequently Asked Questions

General

What is Bob?

Bob is an AI-powered coding companion that can help you write code, refactor existing code, and complete various tasks. Bob's capabilities can be extended with custom modes and instructions.

How does Bob work?

Bob uses large language models (LLMs) to understand your requests and translate them into actions. It can:

- Read and write files in your project.
- Run commands in your terminal.
- Browse the internet (if enabled).
- Use external tools via the Model Context Protocol (MCP).

You interact with Bob through a chat interface, where you provide instructions and review/approve its proposed actions.

What can Bob do?

Bob can help with a variety of coding tasks, including:

- Generating code from natural language descriptions.
- Refactoring existing code.
- Fixing bugs.
- Writing documentation.
- Explaining code.
- Answering questions about your codebase.
- Automating repetitive tasks.
- Creating new files and projects.

Are there any restrictions on processing certain data types with Bob?

The following data types should **not** be processed with Bob:

- SPI

- Personal Health Information (PHI)
- Export Regulated (ITAR, APP etc)
- Regulated Financial Data (Banking numbers, PANs, non-public financial data)

What are the risks of using Bob?

Bob is a powerful tool, and it's important to use it responsibly. Remember:

- **Bob can make mistakes.** Always review Bob's proposed changes carefully before approving them.
- **Bob can execute commands.** Be cautious when asking Bob to run commands, especially if you're using auto-approval.

Usage

How do I start a new task?

Open the Bob panel and type your task in the chat box. Be clear and specific about what you want Bob to do.

How do I switch between modes?

Use the dropdown menu in the chat input area to select a different mode, or use the `/` command to switch to a specific mode.

What are tools and how do I use them?

Tools are how Bob interacts with your system. Bob automatically selects and uses the appropriate tools to complete your tasks. You don't need to call tools directly. You will be prompted to approve or reject each tool use.

What are context mentions?

Context mentions are a way to provide Bob with specific information about your project, such as files, folders, or problems. Use the "@" symbol followed by the item you want to mention (e.g., `@/src/file.ts`, `@problems`).

Can Bob run commands in my terminal?

Yes, Bob can run commands. You will be prompted to approve each command, unless you've enabled auto-approval for commands. Be extremely cautious auto-approving commands.

How do I provide feedback to Bob?

You can provide feedback by approving or rejecting Bob's proposed actions. You can provide additional feedback by using the feedback field.

Can I customize Bob's behavior?

Yes, you can customize Bob in several ways:

- **Custom instructions:** Provide general instructions that apply to all modes, or mode-specific instructions.
- **Custom modes:** Create your own modes with tailored prompts and some tool permissions.
- **Bob rules:** Create markdown rule files in your `.bob/rules/` folder in the root of project to provide additional guidelines.
- **Settings:** Adjust various settings, such as auto-approval, diff editing, and more.

Does Bob have any auto approval settings?

Yes, Bob has a few settings that when enabled will automatically approve actions.

Advanced Features

What is MCP (Model Context Protocol)?

MCP is a protocol that allows Bob to communicate with external servers, extending its capabilities with custom tools and resources.

Can I create my own MCP servers?

Yes, you can create your own MCP servers to add custom functionality to Bob.

Troubleshooting

Bob isn't responding. What should I do?

- Make sure your API key is correct and hasn't expired.
- Check your internet connection.
- Try restarting VS Code.

Bob made changes I didn't want. How do I undo them?

Bob uses VS Code's built-in file editing capabilities. You can use the standard "Undo" command (Ctrl/Cmd + Z) to revert changes. You can also use checkpoints to revert changes made to a file.