OPSIVE

Atlas

Complete Product Documentation

Generated from the documentation source on 2026-10-16
https://opsive.com/support/documentation/atlas/

Contents

Atlas

- Quick Start
- Installation and Updates
- Account and Product Access
- Models and Providers
- The Atlas Window
- Asking Questions
- Providing Context
- Planning and Package Builds
- Generating and Applying Code
- Modify, Review, and Test
- Product Guides
  - Behavior Designer Pro
  - State Designer
  - Ultimate Character Controller
- Project Instructions
- Privacy and Project Data
- Troubleshooting and FAQ
- Commands and Shortcuts
- AI Assistants and Skills
- Credits and Billing

Atlas

View this page online

Atlas helps you work inside Unity or through a connected AI assistant with Behavior Designer Pro, State Designer, and Ultimate Character Controller Version 3. Ask a product question, explain an existing setup, plan a feature, or generate and revise C# source for the product you own.

> Image unavailable in the current public documentation: `atlas-window-overview.png` — The Atlas product header and an illustrative question and answer.

Start here

New to Atlas? Follow Quick start to connect and create a small, testable task. If Atlas is not installed, begin with Installation and updates.

I want to...	Read
Connect my account or verify a purchase	Account and product access
Use prepaid credits, buy a top-up, or manage monthly renewals	Credits and billing
Set up local Codex, Claude Code, or an API key	Models and providers
Connect my existing AI assistant or use Atlas skills	AI assistants and skills
Understand the window	The Atlas window
Explain a graph or diagnose a problem	Asking questions and Providing context
Design a feature before writing code	Planning and package builds
Create or change scripts	Generating and applying code and Modify, review, and test
Apply my project's conventions	Project instructions
Understand what leaves my computer	Privacy and project data
Fix a connection or request problem	Troubleshooting
Look up a command	Commands and shortcuts

Product guides

Open Product Guides for the Behavior Designer Pro, State Designer, and Ultimate Character Controller walkthroughs.

Atlas access is verified for each product. Installing a supported product and owning it are separate requirements. Model choices also depend on your client, account, and configured provider.

A useful working loop

1. Explain the outcome you want and attach the relevant context.
2. Ask for an explanation or plan when the behavior is uncertain.
3. Review the proposed source and its destination before applying it.
4. Let Unity compile, then test the behavior in a small scene or graph.
5. Return the exact error or unexpected result in the same chat.

Atlas validates generated source, but a valid file is not proof that a gameplay feature works. You still assign references, configure graphs or abilities, and test the result. Generated node source does not automatically assemble an entire behavior tree or state machine.

The examples in this manual illustrate the workflow; model responses and available model names will vary. Download Atlas from the Atlas product page.

Quick Start

View this page online

This walkthrough creates a small Behavior Designer Pro task that stays active until interrupted. It keeps the first result easy to inspect and test. For a different product, complete the same connection steps and then use the State Designer or Character Controller example.

Before you start

Use Unity 2022.3 or newer, a compatible supported Opsive product, Atlas, and a configured model provider. Resolve existing Unity Console compilation errors first. Keep your project in source control so you can compare the resulting changes.

Choose how to pay for model usage: your local Codex or Claude Code account, your own API key, or Atlas credits when purchase options are available. Credits require no separate provider subscription; you can buy a one-time pack without a monthly plan.

Connect

1. Open Atlas from your product's menu or docked panel. See Installation for the product menus.
2. Choose **Sign in to Atlas** and complete the one-time connection in your browser.
3. Use the Opsive account that owns the product. Enter a matching Unity invoice only if the browser requests one.
4. Return to Unity and confirm that the relevant product and a usable model are selected.

See Account and product access if the connection does not complete.

Ask, then generate

Start with an explanation:

```
/ask When should a Behavior Designer Pro Action return Running instead of Success?
```

Then request one concrete file:

```
/gen Create SimpleIdleTask, a GameObject Action for Behavior Designer Pro that returns Running until interrupted. It must not move the character or complete automatically.
```

> Image unavailable in the current public documentation: `atlas-generate-task.png` — An illustrative SimpleIdleTask result with source expanded and file actions available.

Apply and verify

1. Expand the generated file. Confirm its class name, base class, and update result match the request.
2. Choose **Save as...** and save to a project folder such as `Assets/AI/Tasks/SimpleIdleTask.cs`. Confirm the write when prompted.
3. Wait for Unity to compile. If it fails, inspect the exact Console error and the offered correction workflow.
4. Add the task to a small test behavior tree. Assign that tree to a test object using the normal Behavior Designer workflow.
5. Enter Play Mode and confirm the task remains **Running**. Stop or interrupt it and confirm the character has no unintended movement or state changes.

An idle task staying **Running** is the expected result. It is not a task that should progress to the next child in a Sequence by itself.

Use Generating and applying code for file controls and correction behavior. Continue with Providing context when your next request needs your own components or an existing graph.

Installation and Updates

View this page online

Atlas runs inside the Unity Editor. Install it into the project containing the Opsive product you want help with.

Requirements

Requirement	Supported starting point
Unity	Unity 2022.3 or newer
Behavior Designer	Behavior Designer Pro 3.3.0 or newer
State Designer	State Designer 1.3.0 or newer
Character Controller	Ultimate Character Controller 3.3.7 or newer within Version 3
Local Codex	Windows or macOS, with the Codex app or standalone CLI and an eligible signed-in account
Connection	Internet access to Opsive and the selected model provider

Classic Behavior Designer and Character Controller Version 4 are not interchangeable with the product versions described in this manual. The optional Unity native-tool integration requires Unity 6 or newer; ordinary Atlas use does not.

Install Atlas

1. Install or update the supported Opsive product and let Unity finish importing it.
2. Download Atlas from the Atlas product page. Use that page for the standalone download rather than the general Opsive Downloads page.
3. Import the package into the same Unity project and wait for compilation to finish.
4. Open Atlas from your installed Opsive product using the menu listed below.
5. Complete account connection and provider setup.

Use the menu for the product you want help with:

Product	Menu
Behavior Designer Pro	**Tools > Opsive > Behavior Designer > Atlas**
State Designer	**Tools > Opsive > State Designer > Atlas**
Ultimate Character Controller	**Tools > Opsive > Ultimate Character Controller > Atlas**

Behavior Designer Pro and State Designer can also host Atlas as a docked panel. A product panel supplies its product context automatically.

Update an existing installation

Commit or back up the project, let active Atlas requests finish, and import the newer Atlas package. If an update is advertised in the window, open **Settings > Download Atlas update...**. The item is disabled when no update is offered.

Avoid keeping two imported copies of Atlas under different folders. Duplicate assemblies or types can prevent compilation and make menu items disappear. After updating, wait for compilation, reopen Atlas, and check the active product and provider.

Atlas keeps project-local chat state across ordinary window closures and assembly reloads. That state is not a substitute for saving generated source or plans. See The Atlas window.

If a menu is missing, check the Unity Console and resolve any compilation errors. If the Atlas panel still shows **Download Atlas** after importing, restart Unity and reopen the panel. If the prompt remains with no Console errors, contact support with your Opsive product version, the Atlas package filename, and screenshots of the imported Atlas files and Console. See Troubleshooting.

Account and Product Access

View this page online

Atlas uses your Opsive account to verify which products it can assist with. Your model-provider login is separate: connecting an Opsive account does not sign you in to Codex or configure an API key.

Connect Unity to your account

1. Open Atlas from the product you want to use. See Installation for the product menus.
2. Choose **Sign in to Atlas**.
3. Complete the one-time browser connection with the correct Opsive account.
4. If requested, enter the Unity Asset Store invoice for that product.
5. Return to Unity. Atlas completes the connection automatically after approval.

> Image unavailable in the current public documentation: `atlas-sign-in.png` — Atlas's initial sign-in view before opening the one-time browser connection.

The browser checks direct Opsive purchases and invoices already linked to your account. A newly verified invoice is associated with that account. You do not need to copy an Atlas access token into Unity.

Always start the connection from Unity. A bookmarked connection URL may have expired or belong to an earlier attempt. If approval times out, start a new sign-in attempt.

Work with more than one product

Product access is checked separately. Owning State Designer does not unlock Behavior Designer Pro. A product also needs to be installed and compiled in the current project before you can use its Atlas integration.

Changing the active product starts a separate chat. Keep an implementation in the chat for the product it targets, especially when continuing a plan or modifying an earlier result.

Ultimate Character Controller, UFPS, and Third Person Controller use the controller-family integration. A paid add-on still needs to be installed for Atlas to use its add-on-specific catalog context.

Credits and subscriptions

Open Account > Atlas credits & devices to see available credits, pending usage, purchase options, and monthly renewal preferences. The account dashboard also links to this page when the Atlas account service is installed.

Prepaid credits let Opsive handle model billing without a separate provider subscription or API key. Choose an optional $5 monthly refill of 40,000 credits, or one-time packs of 40,000 for $5, 85,000 for $10, and

220,000 for $25. Prices are USD before applicable taxes. Purchases are available only when listed on the account page.

Paid credits roll over without expiry and remain available after subscription cancellation. You can opt to skip a monthly renewal when at least 40,000 credits remain available. See Credits and billing for costs, pending usage, cancellation, and refunds. Buying credits does not unlock an Opsive product you do not own.

Devices and sign-out

Open **Settings > Account and devices...** to inspect the connection and device controls. Connected devices are also available through your Opsive account. Disconnect a device you no longer use; it must reconnect before making further authenticated requests.

Settings > Sign out ends the Atlas connection in Unity. It does not sign out your independent Codex installation or revoke an API key at its provider. See Privacy and project data for the distinction between account access, local chats, and server records.

If purchase verification fails

Check the signed-in Opsive account and the selected product first. Use the invoice for that product, not a different asset or add-on. If verification still fails, contact Opsive support with the product and verification error. Do not post invoice details, one-time connection links, or credentials in a public forum.

Models and Providers

View this page online

Atlas assembles product and project context, then uses your selected model to answer or generate source. Configure your connection in **Atlas Providers**, then choose the active provider and model from the chat dropdown. Changing the **Provider** dropdown in the settings window only changes which settings are displayed.

Choose a provider route

Route	Setup	Usage
Codex	Codex app or standalone CLI on Windows or macOS, signed in with the account you intend to use	Uses that local Codex account's access and limits
Claude	Your own Claude Code installation, including a supported desktop-app runtime, signed in separately	Uses that runtime account's access, limits and billing
Codex API	Add your OpenAI API key in Atlas	Provider API billing applies
Claude API	Add your Anthropic API key in Atlas	Provider API billing applies
Atlas credits	Add prepaid credits to your Opsive account and use an available API model without a personal key saved for that provider	Opsive handles API billing and deducts usage from your available credits

Current Atlas services allow every signed-in account to configure its own OpenAI or Claude key, without a separate enablement step. An older service may still return a local-Codex-only catalog. No Opsive subscription is required for local Codex, Claude Code, your own API key, or one-time credit packs.

Use Atlas credits

Credits are an optional way to use models without managing a provider subscription or API key. The monthly option adds 40,000 credits for $5 USD; one-time packs add 40,000 for $5, 85,000 for $10, or 220,000 for $25, before applicable taxes. Paid credits roll over and remain usable after cancellation. Purchase availability is shown in your Atlas account.

A saved personal key takes precedence for its API provider. Without one, an available managed API model uses your Atlas credit balance. Local Codex and Claude Code always use their own runtime accounts and do not spend Atlas credits. Follow Credits and billing for setup and switching from a saved personal key.

Set up local Codex

1. Install and sign in to the Codex app or standalone CLI on the same computer as Unity. Follow OpenAI's authentication instructions.
2. Open **Settings > AI providers and API keys...** and set **Provider** to **Codex**.

3. Set **Run with** to **Auto (CLI, then desktop app)**, **Codex desktop app**, or **Codex CLI**.
4. For a standalone CLI that is not found automatically, set **CLI executable** to its executable path.
5. Choose **Check Runtime & Sign-in** and resolve any reported installation or login issue.
6. Return to the chat and choose an available local Codex model from the model dropdown.

The automatic route tries the standalone CLI before the desktop app. Choose a route explicitly if more than one installation exists and you want a particular one. Atlas does not need you to paste Codex login tokens into Unity.

OpenAI distinguishes ChatGPT sign-in from API-key authentication. Use the intended account and authentication method; a ChatGPT subscription and separately billed API usage are different routes.

Set up Claude Code

Atlas can run your own Claude Code installation on Windows or macOS. You do not need to add an Anthropic API key to Atlas for this route. Claude Code uses its own configured sign-in method, which may have subscription or provider usage limits and charges.

1. Install Claude Code, or use a supported Claude desktop app installation that includes the Claude Code runtime.
2. Open **Settings > AI providers and API keys...**, then select **Claude** in **Atlas Providers**.
3. Set **Run with** to **Auto**, **Claude desktop app**, or **Claude CLI**. If the standalone executable is not discovered, set **CLI executable** to its path.
4. Choose **Check Runtime & Sign-in**. If sign-in is required, choose **Sign In to Claude Code** and complete Claude Code's own sign-in in the terminal window it opens.
5. Check the status again, return to the chat, and select an available Claude Code model. Viewing provider settings alone does not switch the active chat model.

Signing in to the Claude desktop app does not automatically sign in the Claude Code runtime it installs. The runtime has a separate sign-in; **Sign In to Claude Code** launches that flow without asking you to copy credentials into Atlas.

Atlas runs the installed program without copying it and does not read or upload its credentials. These local runtime calls do not consume Atlas model credits. The account used by Claude Code determines available models, limits and billing. **Claude** and **Claude API** are separate choices: the latter uses an Anthropic API key stored by Atlas.

Add an API key

Create a key with your provider

Create the key in the provider account you want to use for Atlas:

Atlas provider	Where to create the key
Codex API	Sign in to OpenAI API keys, choose the intended project, and create an API key. OpenAI's developer quickstart explains this setup.
Claude API	Sign in to the Claude Console API keys page, choose **Create key**, and select the account and workspace the key should use. See Anthropic's API key instructions.

For Claude, a key scoped to one workspace can be used without entering a workspace ID in Atlas. If the key is not scoped to a workspace, copy that workspace's ID from the Console and enter it in Atlas's **Workspace ID (optional)** field. Anthropic's workspace selection instructions explain which keys require it.

Your provider account controls API access, available models, billing, and usage limits. A saved key must have access to the model you select; successful key validation does not guarantee that the account has enough quota for generation.

Save the key in Atlas

1. Sign in to Atlas, then open **Settings > AI providers and API keys...** in the Atlas window.
2. In **Atlas Providers**, set **Provider** to **Codex API** or **Claude API**, matching the account that issued your key.
3. Paste the complete key into **New API key**. For Claude, also enter **Workspace ID (optional)** when your key requires it.
4. Choose **Save Key**, or **Replace Key** if that provider already has a saved key.
5. Wait for **API key validated and saved. The model list has been refreshed.** The provider settings then show **Using your key** with a masked identifier. Atlas checks the key before replacing an existing credential; if validation fails, the previous key is kept.
6. Return to the chat and select a model from that API provider in the model dropdown. Saving a key or viewing its settings does not switch the active chat model.
7. Send a short question to confirm the selected model can answer with your key. If the provider reports a quota or billing error, check that provider account's API usage allowance.

This setup uses the Atlas settings window; you do not need to put the key in your Unity project files or set an environment variable for Atlas.

Replace or remove a key

To replace a key, open the same provider settings, enter the new value in **New API key**, and choose **Replace Key**. To stop Atlas using the saved key, choose **Remove Key** and confirm the removal. Use **Refresh** to retrieve the current saved-key status.

Adding a key does not remove the local Codex or Claude Code options. Atlas sends the key over HTTPS and stores it encrypted on the server; Unity does not retain the entered key. Removing it from Atlas does not revoke it at OpenAI or Anthropic. Manage or revoke the provider key in that provider's dashboard when needed.

Never attach a key to a chat or include it in project instructions. Provider limits, available models, and billing are controlled by the provider account.

Model and effort

The chat model picker selects both provider and model. Atlas initially prefers local Codex when it finds a local runtime, and retains an explicit provider choice. If that route becomes unavailable, check its settings and choose an available model; Atlas does not silently replace an explicit choice with another provider.

The model picker shows choices returned for the current connection. Effort choices vary by model; higher effort may take longer and consume more provider resources. Start with the default, then increase it for a difficult design or debugging task if needed. A model name visible in an example image is not a promise of availability for every account.

Use `/model` and `/effort` for keyboard selection. See Commands and shortcuts, Privacy, and Troubleshooting.

Unity project tools

Local Codex can inspect the live Editor on Unity 6 or newer when Unity CLI and Unity Pipeline are installed. Inspection is enabled by default when those dependencies are available. Scene editing has a separate per-project opt-in and requires an explicit request to change the project. See Providing context for setup, supported edits, and limitations.

Connect an assistant to Atlas

You can also use Atlas from an external AI assistant through the Atlas plugin. Its tools and skills provide project inspection, product knowledge, validation and preview/apply workflows. This is a separate connection from choosing the model inside the Atlas window. See AI assistants and skills.

The Atlas Window

View this page online

The Atlas window keeps your conversation, project context, and generated results together. Open it from a supported product's menu or docked panel.

> Image unavailable in the current public documentation: `atlas-window-overview.png` — The Atlas product header and an illustrative question and answer.

The numbered callouts identify **1** the product header and **2** the conversation. The composer and request controls sit below the conversation.

Main controls

Area	Purpose
Product header	Shows the product context supplied by the product integration.
Chat tabs and **All chats**	Return to earlier work or start a separate conversation.
Conversation	Displays your requests, answers, plans, progress, and generated file cards.
Composer	Enter a request or a slash command. Use **Shift+Enter** for a new line.
Attachment chips	Show context pinned to this chat. Remove a chip with its × control.
Paperclip	Attach files supported by the selected provider and model. Drag-and-drop and clipboard screenshot paste are also supported.
Model and effort controls	Choose the active provider, model, and reasoning effort. Provider settings do not switch this selection.
Settings button	Account, provider setup, project instructions, file-access settings, updates, and sign-out.

Use `/ask`, `/plan`, or `/gen` to make your intention explicit. A natural-language request can also express that intent. See Commands and shortcuts for the full list.

Keep conversations focused

Continue in the same chat when asking about the previous answer, correcting a file, or building a plan. Start a new chat for an unrelated feature. Changing products creates a separate chat so a result is not continued under a different product API.

Attachments stay with their chat and are included again when you send another request. Remove stale context before changing the subject. A Unity object attachment is read again at send time; the snapshot is

not a continuous recording of the scene.

Read a result

Expand a file to review the source before choosing its file action. A multi-file result has individual file cards and an **Apply all** action when the complete result is ready. A saved plan has its own **Save plan...** action.

Show work exposes request progress when available. It is useful for understanding whether Atlas is collecting context, generating, or validating; it does not replace reading the final result.

For supported failures, Atlas offers **Retry request** or **Resume package build**. Let Unity finish compiling or reloading before using them.

Account and credits

Use **Settings > Account and devices...** to refresh your Atlas credit balance and pending usage. **Buy credits / manage subscription...** opens your website account, where available packs and renewal settings are shown. Paid credits roll over and remain usable after cancellation. See Credits and billing for setup and costs.

Persistence

Atlas saves bounded, project-local chat state across window closures and domain reloads. It is not an unlimited archive. Save useful plans as Markdown and generated code as ordinary project files. Clearing a chat removes its local working conversation; it is not a server-data deletion request.

Continue with Providing context or Generating and applying code.

Asking Questions

View this page online

Use **Ask** when you want an explanation, diagnosis, or recommendation before creating code. Start a request with `/ask` to make that intention explicit.

```
/ask Explain when this behavior tree returns to Patrol after the target is lost. Do not generate code.
```

> Image unavailable in the current public documentation: `atlas-ask-explanation.png` — An example Atlas answer explaining the Running and Success task states.

Ask about an existing setup

1. Open Atlas for the relevant product.
2. Attach the object, graph, or script involved. See Providing context.
3. State what you expect and what actually happens.
4. Include the shortest steps that reproduce the difference.
5. Ask Atlas to identify which setting or lifecycle step explains it.

A useful request is specific:

```
/ask This guard switches from Patrol to Chase when the player is visible, but never returns to Patrol. I expect it to return after losing sight for two seconds. Explain which transition or condition to inspect first.
```

An attached graph gives Atlas topology and serialized node settings. It does not prove what happened in every frame of a previous Play Mode session. Include the active state, task status, Console error, or screenshot that demonstrates the problem.

Refine the answer

Keep follow-ups in the same chat. If Atlas assumes the wrong component or lifecycle, name the exact type and attach the relevant source. Ask it to separate observations from assumptions and to propose one check at a time.

For a broad feature request, ask whether a built-in task, Action, Condition, or ability already satisfies it. Using the existing implementation can avoid unnecessary custom code.

Move from explanation to implementation

When the approach is clear, use `/plan` to define responsibilities and verification steps, or `/gen` for a small, concrete artifact. Ask mode itself is useful without creating or saving a script.

If the issue persists, share the installed versions, a minimal reproduction, and the exact error with Opsive support. Review Troubleshooting before sending a report.

Providing Context

View this page online

Give Atlas the part of the project that explains your request: the affected object, relevant script, graph, screenshot, or supported file. Focused context makes an answer easier to check.

> Image unavailable in the current public documentation: `atlas-attached-context.png` — Atlas composer with Unity context and reproduction.md attached, plus the paperclip and model controls.

Attach Unity objects and scripts

Drag an object from the **Hierarchy** or an asset from the **Project** window into the composer. Confirm its chip appears before sending. Atlas supports up to five context attachments per chat, including files and images.

Attachment	What it contributes
GameObject or component	A bounded snapshot of hierarchy, components, serialized values, and relevant references
User-authored script	Source context for the question or change
Behavior Designer Pro tree	The corresponding authoring-layer graph, connections, and serialized task settings
State Designer state machine	The corresponding authoring-layer graph, states, transitions, and serialized node settings
PNG or JPEG	Screenshot pixels alongside the text and object context

Large attachments are bounded. An attachment is not a guarantee that every descendant, field, or graph node fits in the request. If a detail is missing, attach a smaller subject and name the relevant type or setting.

Add a screenshot

Use the **paperclip** button, drag a PNG/JPEG into the composer, or paste a screenshot with **Ctrl+V** on Windows or **Cmd+V** on macOS. Crop large images to the relevant inspector, graph, or error before attaching them. Atlas may resize large images; readable text matters more than including the entire desktop.

Describe what the image should demonstrate. For example: “The transition is selected; the condition is true, but the active state never changes.” Remove account information or unrelated private material from the screenshot before sending it.

Attach a file

Choose the provider and model in the chat dropdown first, then use the **paperclip** button to open **Attach file**. The picker filters extensions for that route. You can also drag supported files into the composer. Confirm each attachment chip before sending.

File type	Supported route
UTF-8 text, including source, Markdown, JSON, CSV, YAML, and logs	All routes
PNG/JPEG screenshots	API routes and local Codex models that accept images
WebP images and PDF documents	Codex API, Claude API, and Gemini when offered by the service
Word, PowerPoint, Excel, RTF, and ODT documents	Codex API
Supported audio and video files	Gemini when offered by the service

Gemini availability depends on the service's model catalog; the key settings currently offer OpenAI and Claude. A supported extension is not a guarantee that every model can interpret all content in that format. For a spreadsheet calculation or complex document layout, explain which content matters and check the result.

Nonempty text files must be valid UTF-8, contain no null characters, and be no larger than 128 KB. Documents and other binary files have a 6 MB per-file limit. Encoded media across the request must fit within 8 MB, so several individually valid files may exceed the combined limit. Text and object context is also bounded and may be truncated.

If you change the model after attaching files, Atlas checks compatibility again before sending. Remove an incompatible chip or choose a compatible model. If a cached file is missing, attach the original again.

Context follows the chat

Attachment chips remain attached for subsequent requests in that chat. Use × to remove one. Atlas rebuilds Unity object snapshots immediately before sending, so changing a serialized value can affect the next request without reattaching the object. Files chosen with the paperclip are cached snapshots: remove and reattach a file to include later edits to the original.

Select scripts in the **Project** window when using `/modify`, `/review`, or `/tests`. A selected edit target and a context attachment serve different purposes: the target determines which source is being changed; attachments provide supporting information.

Automatic project context

Atlas also collects project API information, selected examples, installed-product metadata, and applicable project instructions. Context is not restricted to the files you attach manually. Read Privacy and project data before using Atlas with confidential project material.

Optional live Unity inspection

On Unity 6 or newer, open **Settings > AI providers and API keys...**, choose **Codex**, and use **Use Unity project tools when available (Unity 6+)**. Install Unity CLI and Unity Pipeline in the target project first.

Inspection is on by default when the dependencies are available; ordinary prompts still work without them.

Atlas exposes a limited set of live inspection tools for hierarchy, selection, assets, serialized values, settings, and performance information. The integration is pinned to the current Unity project. Turn the setting off to stop using live tools. An explicit attempt to enable it reports a missing Unity version, CLI, or Pipeline dependency.

Optional scene editing

Scene editing is off by default and is separate from inspection. With project tools enabled, select **Allow Unity scene edits on explicit requests** in the same settings window. This preference is saved per project. Then state the change directly, for example: “Create an empty GameObject named PatrolPoints at the origin in this scene.”

The permitted scene operations are creating, renaming, deleting, and transforming GameObjects, adding components, and saving the scene. Atlas is instructed to check Editor status, avoid editing in Play Mode, and delete objects only when you explicitly request removal. Requested edits may be made and saved without another confirmation, so inspect the resulting hierarchy, components, and saved scene.

Turn off the scene-editing option to return to inspection only. Turning off project tools also turns off the editing opt-in. **Access** and `/access` control generated-file confirmations; they do not enable or disable this separate scene-editing setting.

Project tools do not provide arbitrary code evaluation, general-purpose file access, test runs, builds, or Play Mode control. Script changes still use Atlas's validated generation and application workflow. These bounded tools do not automatically author an entire product graph.

Planning and Package Builds

View this page online

Use a plan when a request has several responsibilities, depends on your own APIs, or needs more than one file. A plan lets you review the design before asking Atlas to implement it.

Create and refine a plan

```
/plan Build a reusable door interaction for my guard AI. Prefer built-in movement tasks. Use my DoorController API for opening and closing. List the files, inspector references, and interruption tests before generating code.
```

> Image unavailable in the current public documentation: `atlas-feature-plan.png` — An illustrative door-interaction plan with the Save plan action.

Attach the existing controller and relevant graph. Check the plan for:

- Responsibilities that can use built-in objects.
- Existing project APIs that should be reused.
- Which component owns movement, animation, input, and temporary state.
- Files to create or modify, and their intended destinations.
- Required inspector assignments and a concrete test for each behavior.

Revise the plan in the same chat. For example, “The controller already owns animation; remove animation handling from the tasks.” Choose **Save plan...** to keep a Markdown copy inside the project.

Build the plan

When the plan is ready, say:

```
Build the plan.
```

Atlas uses the latest plan in the conversation to create a dependency-ordered set of C# files and generates it in batches. A larger plan is not limited to the ordinary selected-file modification count. Keep the scope manageable so you can review and test the complete feature.

> Image unavailable in the current public documentation: `atlas-package-files.png` — An illustrative complete multi-file result ready for destination selection and review.

Incomplete package batches stay staged until the complete manifest is ready. When it finishes, inspect every file and choose an `Assets/...` package destination. **Apply all** preserves the planned responsibility folders. Configuration of your scene, graph, or character remains a separate step unless the result explicitly provides a supported workflow for it.

Resume interrupted work

> Image unavailable in the current public documentation: `atlas-resume-package.png` — An interrupted example package with a Resume package build action.

If **Resume package build** appears, wait until Unity is ready and use it to continue the existing package. Avoid creating a second chat with the same build request while the first result is still recoverable.

After a completed apply, let Unity compile and test the dependency chain from the smallest component upward. If writing a batch fails, Atlas attempts to restore earlier file contents; inspect the project and error before retrying. Source control remains the reliable way to review and restore your project.

See Generating and applying code for destinations and access settings, and Project instructions for folder-specific conventions.

Generating and Applying Code

View this page online

Use `/gen` for a concrete implementation request. Name the product, object type, expected behavior, dependencies, and what should happen when a reference is missing or the work is interrupted.

```
/gen Create a GameObject Action for Behavior Designer Pro that calls DoorController.Open. Use the existing
DoorController API, fail when the reference is missing, and do not control character movement.
```

Attach the existing API when Atlas needs to call your code. Atlas prefers existing built-ins and project APIs; a request may produce an explanation that an existing object already does the job.

Review the result

> Image unavailable in the current public documentation: `atlas-generate-task.png` — An example generated task with its source expanded before saving.

Check the source, target product, class and file names, lifecycle methods, references, and output destination. Structural validation happens before the preview, while Unity compilation happens after a file is saved. Neither check verifies the complete gameplay behavior.

Action	Use it to
Apply or the file's save action	Write the reviewed result to its intended destination, following the current access policy
Save as...	Choose a different project filename or destination
Copy	Copy source for manual use
Discard	Dismiss a proposed result; this is not an undo of an already saved file
Apply all	Apply a complete multi-file result after reviewing its destinations

Save generated C# under `Assets/...`. Do not target installed package or Opsive vendor source. For a modification, Atlas maps the result back to the selected original source path. A plan build instead asks for a package root and preserves its planned subfolders.

File-access settings

Open **Settings > Access**, or type `/access`:

Setting	Confirmation behavior
Ask for approval	Confirm every file write
Approve for me	Confirm overwrites of existing files
Full access	Apply supported file actions without confirmation

> Image unavailable in the current public documentation: `atlas-access-options.png` — The access command showing Atlas's three file-confirmation policies.

These choices control confirmation for generated files and preserve destination validation. They are separate from **Allow Unity scene edits on explicit requests**, which controls bounded scene tools for local Codex. See Optional scene editing. Start with **Ask for approval** while learning the file workflow.

Compilation and correction

After saving, wait for Unity to compile. If a generated file fails, Atlas collects the real compiler diagnostics and can offer correction. Under the approval policy it asks before starting a correction request. Correction consumes provider resources just like other generation work, and automatic attempts are bounded.

Review the corrected result and the original diagnostic. If an unrelated project error is the cause, fix that error rather than repeatedly regenerating the task. If compilation succeeds, test required assignments, success/failure paths, interruption, and cleanup in Play Mode.

See Modify, review, and test, Planning, and the relevant product guide for the next step.

Modify, Review, and Test

View this page online

Atlas can work on existing user-authored scripts for the active product. Choose **Modify** for a proposed source change, **Review** for feedback, or **Tests** for test source.

Select the target

Select the script in Unity's **Project** window before sending the command. Use a compatible user-authored task, State Designer object, or controller ability. Installed package and Opsive vendor source are excluded from these editing workflows.

For a coordinated modification, select two to eight compatible user scripts. Keep the active product consistent with those scripts. Supporting attachments can explain the change, but do not replace selecting the intended edit targets.

Modify a script

```
/modify Add a missing-target check. Preserve the existing serialized fields and public API. Explain what changed.
```

> Image unavailable in the current public documentation: `atlas-modify-result.png` — An illustrative modification result awaiting review before writing to the original source path.

Review the returned source against the original. Check public members, serialized field names, lifecycle behavior, and destination paths. Changing a serialized field can affect existing inspector assignments, even if the file compiles.

For a selected-file modification, **Apply** targets the original file. Multi-file results map their filenames back to the selected source paths. Inspect all destinations before **Apply all**. Use **Save as...** only when you deliberately want a separate file and have checked class-name conflicts.

Review without editing

```
/review Check this task for missing references, interruption cleanup, and incorrect success or failure results.
Explain each issue before proposing a change.
```

> Image unavailable in the current public documentation: `atlas-review-checklist.png` — An illustrative review answer focused on missing references and interruption behavior.

A review provides feedback; it does not apply edits. Check each finding against the source and an observable test case. Continue with `/modify` only when you want a proposed implementation.

Generate tests

Select the target script and request focused cases:

```
/tests Cover the missing-target path, normal completion, and interruption. Keep setup deterministic and explain any required scene or object fixtures.
```

Inspect the generated test source and place it in the appropriate test assembly for your project. Use Unity's **Test Runner** to execute the intended EditMode or PlayMode tests. Generated test source is not evidence of a passing run; read the actual results and investigate failures.

For movement, animation, input, or networking, also verify the behavior in a representative scene. A unit test does not automatically exercise the complete character or graph integration.

See Generating and applying code for write controls and compiler correction, and Project instructions for conventions that should apply to modifications and tests.

Product Guides

View this page online

Choose the guide for the Opsive product you are working with. Each guide explains how to give Atlas the right context, request product-specific source, and test the result in Unity.

Choose your product

Product	What the guide covers
Behavior Designer Pro	Explain behavior trees, generate custom tasks, and check task lifecycle behavior.
State Designer	Explain state machines and generate GameObject or ECS Actions and Conditions.
Ultimate Character Controller	Generate and test Version 3 abilities, including start, stop, cleanup, and character integration.

Before you start

Install the supported product in your Unity project and verify access with the Opsive account that owns it. See Account and product access if a product is unavailable in Atlas.

Connect a model using Models and providers, then attach the tree, state machine, character, or relevant source described in your product guide. Review generated files and test them in a small scene before using them in your full project.

Behavior Designer Pro

View this page online

Atlas can explain attached Behavior Designer Pro trees and generate custom Action, Conditional, Decorator, and Composite task source. Specify GameObject or ECS when requesting a task. This guide targets Behavior Designer Pro 3.3.0 or newer, not classic Behavior Designer.

Start with the existing tree

Open **Tools > Opsive > Behavior Designer > Atlas**, or use the product's docked Atlas panel. Attach the tree or its owning object and describe the behavior you want to understand.

```
/ask Explain the order of the branches in this tree and which condition can interrupt Patrol. Check whether a built-in task already provides the behavior I need.
```

Atlas reads the corresponding authoring-layer graph and serialized settings. A source-generation result creates task types; it does not automatically build and wire an entire tree.

Example: an idle Action

1. Use a small test tree and save your current project changes.
2. Request a GameObject Action with one observable responsibility:

```
/gen Create SimpleIdleTask, a GameObject Action that remains Running until interrupted. Do not move the character or change any Animator values.
```

3. Review the task source, save it under `Assets/AI/Tasks`, and wait for compilation.
4. Add it to the test tree through the normal task selection workflow.
5. Run the tree. Confirm the task remains **Running** and can be interrupted as intended.

> Image unavailable in the current public documentation: `atlas-generate-task.png` — An illustrative generated Behavior Designer Pro idle task with its update method visible.

For a movement or interaction task, attach the real controller API and state who owns navigation, rotation, animation, and stopping. Ask for missing-reference handling and cleanup when the task ends.

Review by task type

Type	Important checks
Action	Startup, repeated updates, correct result, and cleanup on interruption
Conditional	The exact condition, null behavior, and repeatability of evaluation
Decorator	How the child result is transformed and when the child can run again
Composite	Child order, termination, reset, and interruption behavior
ECS task	Required components, enableable state, system/job lifecycle, and supported package APIs

Connect and verify

Assign the generated fields and Shared Variables in the task inspector. Test both expected and unexpected paths: missing target, already-completed work, interrupted work, and re-entry. Then add the task to the complete tree and repeat the relevant checks.

Use Creating a GameObject task, Creating an Entity task, and Variables for the underlying framework. See Modify, review, and test for revising your own tasks.

State Designer

View this page online

Atlas can explain attached State Designer state machines and generate GameObject or ECS Actions and Conditions. Use State Designer 1.3.0 or newer and open **Tools > Opsive > State Designer > Atlas**, or its docked panel.

Explain a state machine

Attach the state machine or its owning object. Ask about a specific transition, state, or execution result:

```
/ask Explain why this door remains in Open after the close request. Identify the transition and condition settings to check before changing code.
```

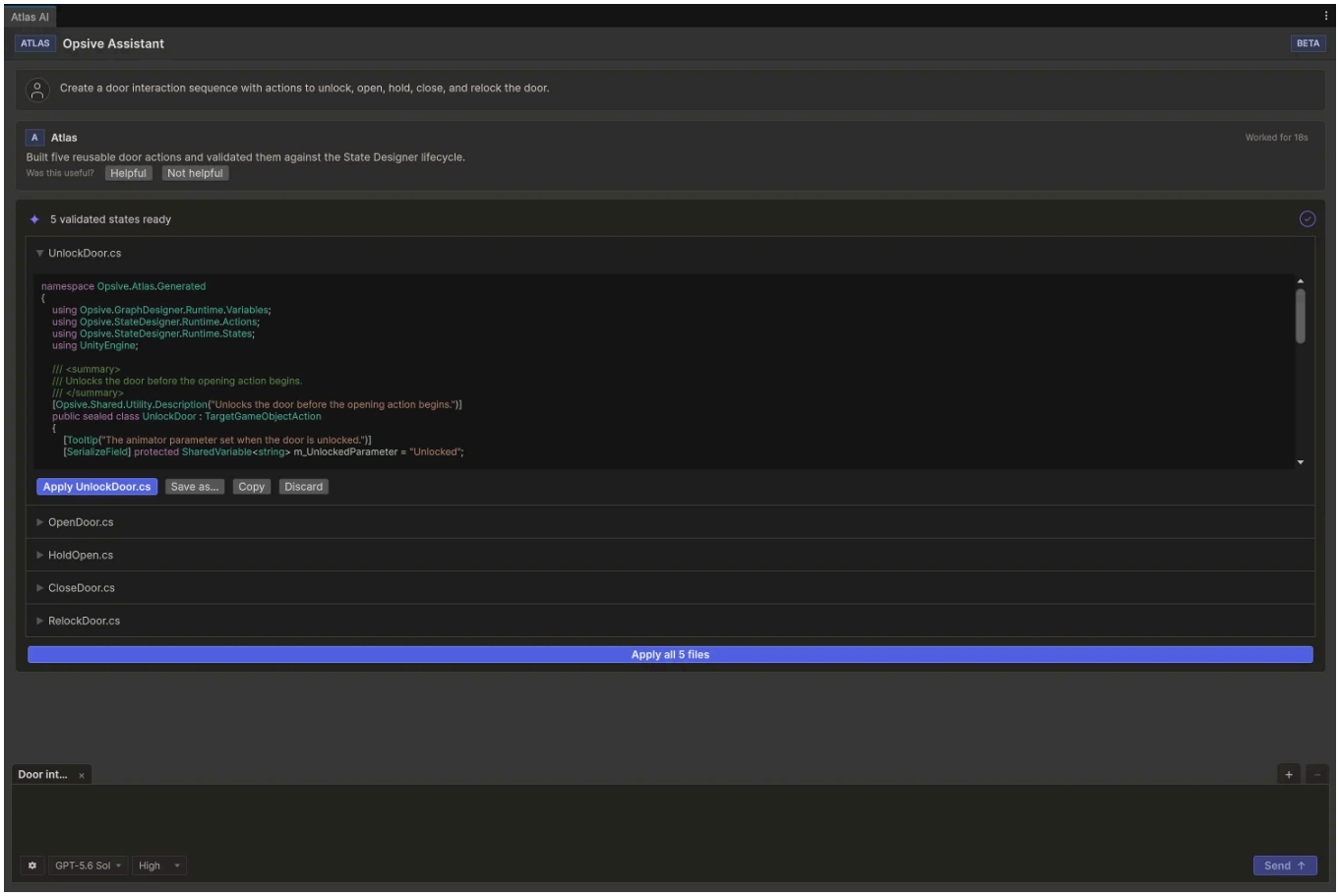
Atlas reads the corresponding authoring-layer graph, including topology and serialized node settings. Include the observed active state or a screenshot when diagnosing a runtime problem. A snapshot does not contain the full history of earlier transitions.

Example: a distance Condition

1. Create a small test state machine with two states and a transition between them.
2. Ask for one condition with explicit missing-reference behavior:

```
/gen Create a GameObject Condition that is true when an assigned target Transform is within a configurable distance of the owner. Return false if the target is missing. Use State Designer's supported Shared Variable fields.
```

3. Review whether the result is a **Condition**, with boolean evaluation, rather than an Action performing work over time.
4. Save the source under an `Assets/...` folder and wait for compilation.
5. Add the Condition to the test transition and assign its owner, target, and distance fields as required by the generated implementation.
6. Move the target inside and outside the threshold. Confirm the transition responds as intended, then remove the target reference and confirm it evaluates false.



Actions, Conditions, and ECS

An Action performs work as part of a state lifecycle. A Condition evaluates whether a requirement is satisfied. State the intended role instead of asking for an ambiguous “node.”

For an Action, test entry, update, completion, exit, interruption, and re-entry. For a Condition, test true, false, and missing-data cases. Avoid moving or mutating unrelated objects as a side effect of a condition check.

For ECS, explicitly request the ECS flavor and describe required component data and scheduling assumptions. One generated source artifact can include the associated authoring, buffer, flag, and system/job types; review the complete artifact.

Finish the integration

Generated source defines new types. Add them to states and transitions through the normal State Designer editor and assign the relevant Shared Variables. A generated package does not automatically assemble your state machine.

Continue with the State Designer manual, State Designer Atlas integration, or Generating and applying code.

Ultimate Character Controller

View this page online

Atlas supports the released Ultimate Character Controller Version 3 family, including UFPS and Third Person Controller. Use version 3.3.7 or newer within Version 3. Version 4 development APIs are outside this guide.

Open **Tools > Opsive > Ultimate Character Controller > Atlas** and use a duplicate test character for a new ability.

Choose the right object

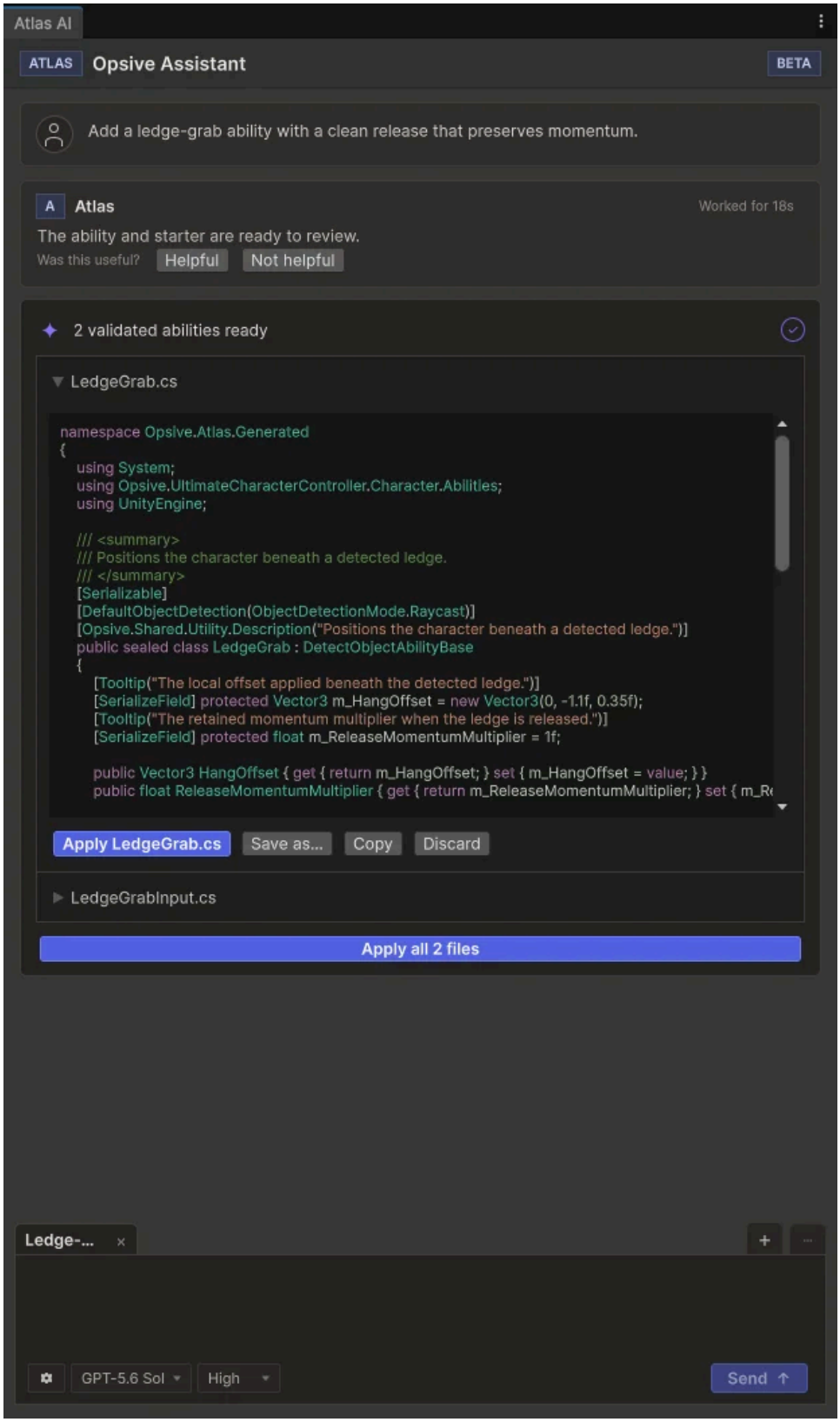
Object	Purpose
Ability	Character behavior coordinated through the ability lifecycle
Item Ability	Item-related behavior using the item-ability framework
Ability Starter	Custom logic for starting an ability
Ability Stopper	Custom logic for stopping an ability

Attach the relevant character, component, and existing user-authored ability source. Explain the active perspective, input setup, Animator requirements, and any installed add-ons the behavior depends on.

Example: plan a short dash

```
/plan Add a short dash ability for my Version 3 character. Use the controller's movement and ability APIs. Define start conditions, stopping, interruption, and cleanup. Do not move the Transform directly. Identify inspector setup and a test plan before generating code.
```

Check how the dash coordinates with existing movement, camera, items, and animation. Define which abilities block it and whether it is concurrent. Only then ask Atlas to build the agreed plan.



Apply and configure

1. Review the generated source and inherited ability lifecycle methods.
2. Save the files under your own `Assets/...` folder and wait for compilation.
3. Add the ability to the test character through the controller's normal **Abilities** inspector workflow.
4. Configure **Start Type**, **Stop Type**, input, ordering, and required fields according to the implementation.

5. Assign any States, Animator parameters, or references the result requires.

Generated source does not configure all character and Animator settings for you. A compiled ability can still fail to start because its start conditions, priority, input, or required components are not configured.

Verify behavior

Test starting, normal stopping, interruption by another ability, disabled character, death/respawn, and perspective changes where relevant. Confirm temporary States, scheduled callbacks, event subscriptions, and motion are cleaned up.

Test any networked behavior through the actual networking integration and authority model. Generating a local ability does not create replication automatically.

Atlas uses installed-product metadata when selecting add-on context. Do not assume an Agility, Climbing, or Swimming API is available merely because another controller-family product is installed.

See Abilities, Creating a New Ability, and the existing Atlas integration guide.

Project Instructions

View this page online

Use `AGENTS.md` files to give Atlas reusable project conventions. Instructions apply to asking, planning, generation, modification, correction, testing, and review, according to the relevant project or file scope.

Add project conventions

Create `AGENTS.md` at the Unity project root, beside `Assets`, `Packages`, and `ProjectSettings`. Keep it concise and actionable:

```
Use the Studio.Gameplay namespace for our runtime scripts.
Use serialized private fields for inspector references.
Prefer existing project components over a second implementation.
Do not rename existing serialized fields without explaining migration.
For gameplay changes, describe missing-reference and interruption tests.
```

Do not put credentials, tokens, or other secrets in this file. Applicable instruction contents are sent with the request.

Folder-specific instructions

Add another `AGENTS.md` inside a narrower folder, such as `Assets/AI/Tasks`, to define rules for that subtree. Atlas reads applicable files from the project root down to the relevant destination or source folder. Deeper files refine conflicting ancestor rules for their own scope.

Generation and planning use the current save folder. Modify, review, tests, and correction use their actual source or saved-file paths. General questions use the project root.

If you save a result into a folder with a different instruction chain, Atlas warns before writing so you can regenerate for the new destination.

> Image unavailable in the current public documentation: `atlas-project-instructions.png` — An illustrative plan referring to project conventions and the intended task folder.

Choose an alternative base file

Open **Settings > Project instructions > Choose AGENTS.md...** to select a project-wide base instruction file inside the Unity project. Automatically discovered nested files continue to refine it.

Choose **Clear chosen file** to return to discovery-only behavior. **Use project AGENTS.md** enables or disables instruction collection; it is enabled by default.

Precedence and limits

Project instructions cannot override Atlas's product rules, response format, or source validation. Within those boundaries, project instructions take precedence over conflicting task-specific preferences, and deeper instruction files take precedence over ancestors.

Atlas accepts up to 16 instruction documents, 32 KB per document, and 64 KB in total. Empty or oversized documents, linked files or directories, and paths outside the project are ignored with a Unity warning.

Natural-language conventions are model guidance, not a guarantee enforced by a compiler. Inspect the result for compliance and keep deterministic formatting or analysis checks in your normal development workflow.

See Privacy and project data and Generating and applying code.

Privacy and Project Data

View this page online

Atlas sends request context to the Atlas service and the selected model route. Running Codex or Claude Code locally keeps that runtime's authentication on your machine, but it does not make Atlas an offline service or keep every prompt and result on your computer.

What can be included

Context	How it is collected
Your prompt and conversation	The current request and retained conversational context
Installed-product information	Product identity, installed version, API information, and relevant catalog context
Project API surface	Exported types from assemblies the scanner includes, with public members and serialized private fields
User-authored source examples	Atlas can automatically include up to five supported script examples, bounded to 8,000 characters each
Selected source	Files selected for modification, review, tests, or correction
Attachments	Bounded object/graph snapshots, text-file content, and supported image, document, audio, or video data
Project instructions	Applicable `AGENTS.md` contents when enabled
Compiler information	Relevant diagnostics during correction and compilation-result information

The client API scan is not initially limited to names that match your prompt. The service selects bounded relevant context for the model. Do not interpret “relevant context” as “only files I manually attached.” Package and Opsive vendor roots are excluded from user-source editing/example workflows; curated product API metadata is a separate input.

Local Codex route

Unity runs Codex using the local installation and its existing authentication. Atlas does not need the Codex login credentials to be uploaded. Atlas still assembles the request, validates results, and stores conversation and generation records. The model interaction also remains subject to the provider account's terms and settings.

When Unity project tools are available, read-only inspection is on by default and Codex may inspect additional live project details. Turn off **Use Unity project tools when available (Unity 6+)** if you do not want live inspection. A separate per-project scene-editing opt-in permits bounded scene changes when you explicitly request them. See Providing context.

Claude Code route

Atlas runs your installed Claude Code program using its own authentication. Atlas does not read or upload that runtime's credentials. The runtime sends model requests using its configured provider account. Atlas still coordinates the request, receives the result, validates generated source, and stores conversation and generation records.

External AI assistants

The optional Atlas plugin can return selected-object and graph snapshots, product API signatures, proposed and original source, and compiler diagnostics to the assistant you connect. That assistant's own model, account and data settings apply to information it receives. Connecting an assistant does not make its work offline.

The Unity connection is disabled until you enable it for the project. The connection file contains a local credential: share its path only with the assistant you intend to connect, never the file contents. Disabling the connection stops access through that bridge. Credential rotation invalidates the previous credential, but a client that can still read the connection file can read the replacement; disable the connection to stop access.

Previews, operation records and original-file journals are stored under the project's **Library** folder. Applying source follows your Atlas access setting and checks whether the file changed after the preview. A successful compile is separate from runtime verification. See AI assistants and skills.

API-key routes

When you configure an OpenAI or Anthropic key, it is sent to Atlas over HTTPS and stored encrypted on the server. The entered key is not retained by Unity. Atlas uses the configured key for that provider's API requests.

Removing the key in Atlas stops Atlas from using that saved key. Revoke it at the provider as well if it should no longer be valid anywhere. Never paste a key into a prompt, screenshot, attached file, or `AGENTS.md`.

Atlas credit route

With prepaid credits, the Atlas service sends the relevant request context to OpenAI or Anthropic using Opsive's configured API credentials. You do not supply a provider key for this route. Atlas records usage against your credit balance alongside the request records described below. Buying credits does not make model requests local or offline. See Credits and billing.

Local and server records

Unity stores bounded chat state in the project's `Library` area so it can survive window closure and domain reload. Images and explicitly selected files also use project-local caches. Selected files are copied into the cache when attached; removing a chip does not delete the original. Generated files you apply are ordinary project files; saved plans are ordinary Markdown files.

Atlas maintains server-side conversation and generation records under its retention configuration. Clearing the local chat, removing an attachment chip, deleting the Unity cache, or signing out does not itself request

deletion of server records or provider records. Removing a chip prevents it from being included in future requests in that chat; it cannot retract an earlier request.

For a data-handling or deletion question, contact Opsive through Support. Provider retention and account settings are separate from Atlas settings.

Before sharing project context

Remove secrets and unrelated personal data from source, instructions, and screenshots. Attach the smallest object or file set needed to explain the issue, and review your organization's rules before sending confidential code. See Account and product access for device management.

Troubleshooting and FAQ

View this page online

Start with Unity's first Console error and the exact Atlas status message. Wait for compilation or a domain reload to finish before retrying a request.

Connection and setup

Symptom	What to check	Next step
Atlas menu is missing	Import completion, duplicate Atlas copies, and Console errors	Resolve the first compilation error, then reopen the editor window. See Installation.
A product is missing	Supported version installed and compiled in this project	Open Atlas from that product's menu and verify its package version.
Sign-in times out	Correct browser account and a current one-time connection	Start a fresh sign-in from Unity; do not reuse an old browser URL.
Purchase cannot be verified	Selected product, owning Opsive account, and matching invoice	Use the owning account or verify the correct invoice. Contact support if it still fails.
Local Codex is unavailable	Local installation, selected runtime route, sign-in, and executable path	Run **Check Runtime & Sign-in** in Atlas provider/runtime settings.
Claude Code is unavailable or signed out	Selected runtime route, installation and runtime sign-in	Choose **Check Runtime & Sign-in**, then **Sign In to Claude Code** if needed. Desktop-app sign-in alone is not sufficient. See Models and providers.
External assistant cannot connect	Matching Atlas package/service/plugin, enabled project connection, copied connection-file path	Reconnect to the intended project after compilation finishes. See AI assistants and skills.
API-key fields are absent	Client version, sign-in, and provider settings view	Update Atlas, sign in, and select **Codex API** or **Claude API** in **Atlas Providers**. Older services may still provide only local Codex models.
Key validation fails	Correct provider, valid key, provider access and billing	Check the provider account, then save the correct key in Atlas's provider settings.
A model or effort is missing	Choices returned for the current provider/account	Select an offered choice; do not assume an example screenshot's model is available.

Unity project tools

Symptom	Next step
Unity inspection cannot be enabled	Use Unity 6 or newer, install Unity CLI and Unity Pipeline in this project, and check **Unity CLI executable**. Ordinary prompts remain available without tools.
An isolated Codex sign-in error appears	Run **Check Runtime & Sign-in** and sign in through the installed Codex runtime. If sign-in cannot be used in the isolated session, turn off Unity project tools to continue without them. Do not paste login credentials into Atlas.
Atlas inspects but will not edit a scene	Enable the separate **Allow Unity scene edits on explicit requests** setting and explicitly describe a supported scene change. Leave Play Mode first.
Changing Provider in settings leaves the chat unchanged	This is expected. Select the active provider and model from the chat model dropdown.

Requests and results

Symptom	What to check	Next step
Request fails or stalls	Network connection, provider limits, Unity reload, and progress text	Let Unity become ready; use **Retry request** when offered.
Package build is incomplete	Whether the result offers **Resume package build**	Resume the existing build rather than applying a partial package.
Attachment is rejected	Selected provider/model, five-attachment limit, file size, UTF-8 text encoding, and cached-file availability	Choose a compatible model, reduce the file, or remove and reattach it. See Providing context.
Atlas describes the wrong graph	Active product, attached owner, and current graph snapshot	Attach the intended graph and state the exact node or transition.
A file cannot be modified	Compatible user-authored target selected in the Project window	Select the intended source; installed package/vendor files are not editable targets.
Saving is blocked	Destination, existing file, access setting, and instruction scope	Choose a valid `Assets/...` destination and review any confirmation or scope warning.
Code fails to compile	Exact diagnostic, product version, and unrelated Console errors	Use the offered correction flow or provide the diagnostic in the same chat.
Code compiles but does not work	Inspector assignments, lifecycle, graph wiring, start conditions, and ownership	Reproduce in a small test setup and report expected versus actual behavior.

> Image unavailable in the current public documentation: `atlas-resume-package.png` — An example interrupted package with the Resume package build action.

Credits and billing

Symptom	Next step
Not enough credits for a request	Check available credits and pending usage in **Settings > Account and devices...**. Add a top-up, let pending calls settle, or choose your own provider account.
Paid credits have not appeared	Refresh the account page. If synchronization does not complete, contact support with the order number. Do not buy the same pack again just to retry synchronization.
Provider billed me despite having Atlas credits	A saved personal API key takes precedence for that provider. See Using credits in Unity to switch to managed access.
Subscription cancelled or renewal failed	Previously paid credits remain available. Cancellation or failure adds no new credits.
Monthly renewal was skipped	The optional skip setting checks whether at least 40,000 credits are available after pending reservations. A skip adds no credits and moves the next payment one month ahead.
Balance is temporarily unavailable	This is a retrieval failure, not a zero balance. Retry later; contact support if it persists.

See Credits and billing for purchase availability, pricing, rollover, and refunds.

Common questions

Does Atlas build the whole graph for me?

Generating task, Action, or Condition source creates types you can add to a graph. You still assemble the graph and configure its references through the product editor. Ask Atlas to explain that setup as part of the plan.

Can I keep using local Codex after adding a key?

Yes. API-key routes and the local Codex route are separate model choices. Check the selected provider before sending a request. See Models and providers.

Does clearing a chat delete server history?

No. It clears the local working conversation. See Privacy and project data for storage and deletion distinctions.

Why did Atlas recommend a built-in object instead of generating a file?

Atlas prefers a supported built-in object when it already satisfies the request. Explain the specific missing behavior if you need a custom implementation.

Send a useful support report

Include the Unity version, Opsive product and version, Atlas package version, operating system, selected provider route, exact error text, and concise reproduction steps. Say whether the failure occurred during sign-

in, generation, application, compilation, or Play Mode. Include a request identifier when Atlas displays one.

Use a small sanitized example or screenshot. Do not send API keys, Codex credentials, one-time connection URLs, or a whole private project by default. The development-only verbose diagnostics menu is not present in ordinary customer builds.

Contact Opsive through the Support page.

Commands and Shortcuts

View this page online

Type `/` in the Atlas composer to see commands. Suggestions show valid arguments for the current connection, including model and effort choices.

> Image unavailable in the current public documentation: `atlas-command-menu.png` — The Atlas command suggestions opened from the composer.

Commands

Command	Purpose	Example
`/ask`	Ask for an explanation without requesting code generation	`/ask Explain this transition.`
`/plan`	Plan an implementation before generating files	`/plan Add a door interaction.`
`/gen`	Explicitly request generation	`/gen Create a GameObject idle Action.`
`/modify`	Modify selected compatible user scripts	`/modify Add a null-target check.`
`/review`	Review a selected script	`/review Check interruption cleanup.`
`/tests`	Generate test source for a selected script	`/tests Cover success and missing-reference cases.`
`/model`	Select an available model	Type `/model` and choose a suggestion.
`/skill`	Select the reasoning effort level	Type `/skill` and choose an offered level.
`/effort`	Alias for `/skill`	Type `/effort` and choose an offered level.
`/access`	Select the file-confirmation policy	`/access ask`, `/access approve`, or `/access full`
`/clear`	Clear the current local working chat	`/clear`
`/help`	Show command and keyboard help	`/help`

In Atlas, `/skill` refers to reasoning effort. It is not a command for installing a plugin or a reusable assistant skill.

Use `/modify` with one script or two to eight compatible scripts selected in Unity's **Project** window. Review and test commands also require an appropriate selected script. Attach additional context separately when it helps explain the task.

Keyboard shortcuts

Shortcut	Action
Enter	Send the request; when suggestions are open, accept the selected suggestion
Shift+Enter	Insert a newline in the composer
Up / Down with suggestions open	Move through suggestions
Tab with suggestions open	Accept the selected suggestion
Escape with suggestions open	Dismiss suggestions
Ctrl/Cmd+Up / Down	Navigate prompt history
Ctrl+V on Windows / **Cmd+V** on macOS	Paste a clipboard screenshot when one is available

Ordinary text paste remains available. Image paste depends on the clipboard containing an image supported by the platform.

Useful settings

- **Account and devices...**: inspect the Atlas connection and device controls.
- **AI providers and API keys...**: configure a provider; choose the active provider and model in the chat dropdown.
- **Use project AGENTS.md**: enable or disable project instruction collection.
- **Project instructions**: choose or clear a project-wide instruction file.
- **Access**: choose when Atlas requests confirmation before generated-file actions. The provider window has a separate opt-in for Unity scene edits.
- **Download Atlas update...**: open an advertised client update when available.

See The Atlas window, Project instructions, and Generating and applying code for the full workflows.

AI Assistants and Skills

View this page online

The Atlas plugin lets a compatible AI assistant use Atlas's product knowledge and work with your Unity project. It includes tools for inspection, API lookup, source validation, previews and compilation feedback, plus three skills that guide common Opsive workflows.

This connection is separate from choosing Codex or Claude Code as the model inside Atlas. Your external assistant can write code itself and ask Atlas to validate it, or delegate generation to the provider selected in the Unity Atlas window.

Download the connector

Download Atlas AI Connector 0.4.0 for Codex and Claude Code.

The ZIP includes the ready-to-run MCP connector, all three skills, a setup helper, and a README with installation and removal instructions. It is a separate download from the Unity `.unitypackage`; you do not need the Atlas source repository or a plugin marketplace listing.

Extract the complete ZIP to a permanent folder such as `C:/Tools/AtlasConnector`. Keep that folder after setup because the assistant runs the connector from there.

Requirements

- Atlas 0.4.0 or newer in Unity, with **Connect an AI assistant** available. Connector 0.4.0 uses the same local bridge protocol introduced in Atlas 0.4.0.
- A supported product installed in the project and authorized on your Opsive account: Behavior Designer Pro, State Designer, or Ultimate Character Controller Version 3, including its supported controller-family variants.
- Codex or Claude Code running locally on the same computer as Unity. In Claude Desktop, use a local **Code** session. Cloud sessions cannot reach the local Unity connection.
- Node.js 22 or newer on the machine running the connector, and Unity 2022.3 or newer for the project connection.

Keep the complete plugin folder, including its bundled `dist/index.js` file. A source checkout needs the connector build described below; copying only the skill files does not provide the connection.

Set up Codex

Open a terminal in the extracted connector folder and run:

```
node setup.mjs codex
```

This registers the `atlas` MCP server for your user account and installs the three skills in `~/.agents/skills`. Restart Codex and start a local session for your Unity project. Codex desktop and CLI share the MCP

configuration on the same host.

Set up Claude Code

Open a terminal in the extracted connector folder and run:

```
node setup.mjs claude
```

This registers the `atlas` MCP server in user scope and installs the three skills in `~/.claude/skills` (or your configured `CLAUDE_CONFIG_DIR/skills`). Restart Claude Desktop and start a local **Code** session for your Unity project, or start a new Claude Code CLI session.

If setup cannot find the client command, use `--cli` followed by the full path to its executable, or use the manual configuration below. Setup preserves an existing `atlas` server and refuses to overwrite edited skills. The bundled README explains upgrades and removal. Add `--dry-run` to preview setup without changing anything.

Connect your Unity project

1. Sign in to Atlas in the Unity project you want the assistant to use.
2. Open **Tools > Opsive > Atlas > Connect an AI assistant**.
3. Turn on **Enable connection** and choose **Copy connection file path**.
4. Complete the Codex or Claude Code setup above, or the manual configuration below.
5. Give the assistant the copied path and ask it to connect using `atlas_connect`. Share the path, not the contents of the connection file.
6. Ask it to check `atlas_get_capabilities` and confirm the project, installed product version and product access before starting work.

Connections are per project. Select the intended objects in Unity before asking about a graph or character setup. Wait for compilation and reloads to finish before inspecting or applying changes. The connector rereads the connection file after reloads so it can reconnect with the current credential.

Account and authentication

An Opsive account and access to the relevant product are still required. Sign in through Atlas in Unity using the normal browser authorization flow. The connector authenticates locally to Unity, and Unity uses its existing Atlas session for server requests and product-access checks. There is no second Opsive login in Codex or Claude Code for this mode. The assistant does not receive your Opsive account tokens. The connection file contains a separate local credential and must remain private.

Manual MCP configuration

Run `node setup.mjs codex --skills-only` or `node setup.mjs claude --skills-only` in the extracted folder. This installs the skills and prints the absolute Node executable and connector paths for your machine.

In Codex, add a STDIO server named `atlas` under **Settings > MCP servers** using those values. For Claude Code, merge the following entry into the `mcpServers` object in `~/.claude.json`, preserving existing settings. Close the client before editing its configuration and restart afterward:

```
{
  "mcpServers": {
    "atlas": {
      "command": "C:/Program Files/nodejs/node.exe",
      "args": ["C:/Tools/AtlasConnector/dist/index.js"]
    }
  }
}
```

Replace both example paths with the values printed by setup. This is an example entry, not a replacement for the whole configuration file. After the server starts, connect the Unity project using the copied connection-file path. The connector does not require Unity CLI.

Use the skills

Skill	Useful requests
`build-opsive-behavior`	Create or modify a task, action, condition or ability; check existing functionality first
`diagnose-opsive-problem`	Investigate a selected graph, transition, character setup or unexpected behavior
`review-opsive-code`	Review existing source against the installed product's APIs and lifecycle

For example: “Use Atlas to explain why this selected State Designer transition never fires,” or “Use Atlas to review this UCC3 ability's start and stop behavior.” Skills guide the workflow; they do not grant product access or permission to apply changes.

Inspect, validate and apply

The assistant can inspect selected-object snapshots and look up the installed product's public and protected API signatures. Documentation searches use the requested product version. Missing exact documentation or contracts remain missing; an older catalog match is identified as approximate and requires explicit opt-in. A Play Mode snapshot is not an execution history.

For a code change, the assistant validates unchanged source and stages a preview with the original and proposed contents. Applying the preview follows the existing Atlas access setting. An intervening file edit makes the preview stale; the assistant must read the new contents and create another preview. The connector cannot grant itself approval.

The apply operation reports compilation status and relevant diagnostics. Generated files are not automatically applied, and a successful compile does not prove the gameplay behavior. Test the result in the intended graph or scene. This connector does not provide arbitrary scene-editing or Play Mode commands.

If generation is delegated to Atlas, it uses the selected Atlas provider unless an explicit model selection is supplied. This can consume that provider's usage. Knowledge lookup and structural validation do not call a

model, although your external assistant's own model use may still incur charges.

Recover or disconnect

If a response is late or interrupted, ask the assistant to check the existing operation instead of starting another generation. Unchanged retries use the same request ID. An interrupted code-generation request may be recovered; an interrupted advice conversation remains unresolved. Applied-file journals preserve original bytes for recovery, but cancelling an operation does not undo files that were already applied.

Turn off **Enable connection** to stop bridge access. The credential-rotation control replaces the local credential; assistants that can still read the connection file can reconnect. Keep connection files private. See Privacy and project data.

Knowledge access without Unity

The connector also supports standalone account access for capabilities, knowledge searches, exact published API contracts and structural validation. It cannot inspect a local project, apply files or report Unity compilation in this mode. Supply the product version explicitly; hosted API contracts must have been published for that exact release.

From the plugin folder, run:

```
node dist/index.js login --api https://atlas.opsive.com --product behavior_designer
node dist/index.js --api https://atlas.opsive.com
```

Use the first command's one-time Opsive authorization link. The second command is the stdio server command to configure in your assistant. For another product, use `state_designer` or `ultimate_character_controller`. The account creates a separately revocable **Atlas AI connector** device. Use `node dist/index.js logout --api https://atlas.opsive.com` to sign it out.

Building the plugin from source

In an Atlas source checkout, run `pnpm install --frozen-lockfile`, then `pnpm agent:package`. The complete plugin is assembled under `plugins/atlas`. Run `node scripts/test-agent-plugin.mjs` to verify the bundle and connection protocol. The repository's `docs/AIAssistantIntegration.md` contains developer setup, API references and release checks. Building a plugin locally does not deploy the matching API service.

Credits and Billing

View this page online

Atlas credits let you use OpenAI or Claude models through Opsive without a separate provider subscription or API key. Atlas itself is included with supported Opsive products; credits pay for AI usage and do not replace product ownership verification.

Credit purchases are available only when purchase options appear in Account > Atlas credits & devices. If no options are shown, sales have not opened or are temporarily unavailable. You can still use your own local provider account or API key.

Choose a credit pack

Option	Price in USD, before applicable taxes	Credits added
Monthly subscription	$5 each month	40,000 per successful payment
One-time top-up	$5	40,000
One-time top-up	$10	85,000
One-time top-up	$25	220,000

Top-ups do not require a subscription. All paid credits go into the same account balance, roll over without expiry, and remain usable after cancellation of a monthly subscription. A top-up does not change your next monthly payment date.

Sign in to the Opsive account you use in Unity before purchasing. Checkout takes payment in USD through Stripe; applicable taxes are shown at checkout. Credit packs are purchased individually and cannot be combined with coupons or other products in the same order.

What a credit pays for

10,000 credits fund $1 of AI provider usage. This is the underlying model usage budget, not the checkout price. Pack prices also include a buffer for payment processing and service costs. The purpose is to offer another way to use Atlas while covering its costs.

For example, 40,000 credits fund $4 of model usage. The number of answers or generated scripts depends on the selected model, input and output size, cached context, and any further model calls. Planning, generation, and repairs all consume usage when they call the model. A complex request can involve several calls; there is no fixed number of requests per pack.

Before a model call, Atlas reserves credits to cover it. **Credits available** excludes these reservations. After usage is confirmed, Atlas deducts the actual charge and releases unused reserved credits. A failed or interrupted request may still have incurred provider usage. If its final cost is uncertain, its reservation remains pending until reconciled.

Use credits in Unity

1. Connect Unity to the same Opsive account used for the purchase.
2. Open **Settings > Account and devices...** and choose **Refresh** to check your available credits. **Buy credits / manage subscription...** opens the account page in your browser.
3. Select an available OpenAI or Claude API model from the chat model dropdown. With no personal key saved for that provider, Atlas uses managed access when it is available to your account.
4. Send your request. Check the account window or website to see the remaining balance and pending usage.

If you have saved a personal API key for the selected provider, Atlas uses that key and the provider bills your account directly. To use credits for that provider, open **Settings > AI providers and API keys...**, select the provider, and choose **Remove Key**. This removes the saved key from Atlas; it does not revoke it with the provider. Local Codex and Claude Code remain separate options and do not spend Atlas credits. See Models and providers.

Monthly renewals and cancellation

The monthly option charges $5 and adds 40,000 credits after each successful payment. Cancel future renewals through **Account > Subscriptions**. Cancellation does not remove credits already paid for. A failed renewal adds no credits; any remaining paid balance is still usable.

On **Account > Atlas credits & devices**, you can enable **Skip my monthly credit payment when I still have at least 40,000 available credits**, then choose **Save preference**. This is optional and is off unless you enable it.

At renewal time, Atlas checks available credits after pending reservations. If at least 40,000 are available, that renewal is skipped: no payment is taken, no credits are added, and the next payment moves one month ahead. If fewer are available, the renewal proceeds normally. If Atlas cannot check the balance, payment is delayed while the check is retried. A top-up can therefore affect whether an opted-in renewal is skipped.

Balances, payments, and refunds

What you see	What it means
Not enough available credits	The next model call needs more than the available balance. Add a pack, wait for pending usage to settle, or choose your own provider account.
Pending usage	Credits are reserved for calls whose final charge is not yet settled. Avoid repeatedly resending an interrupted request.
A paid order but no new credits yet	The payment and balance may still be synchronizing. Refresh the account page. If it persists, contact support with the order number.
Account information unavailable	Atlas could not retrieve the balance. This does not mean the balance is zero. Try again later.
Outstanding credits after a refund	A refund reversed credits that were already spent. The next purchase covers that outstanding balance before adding usable credits.

When a refund is processed, the corresponding proportion of the purchased credits is reversed. Cancelling a subscription stops future renewals; it does not itself refund past payments.

For a billing question or a reservation that remains pending, contact Opsive support with your order number or request identifier. Do not include payment-card details, API keys, or connection links.