Group Milestone 3

Player Enhancements

In this group project, your team will make significant modifications to your automated players. The players will become available over the network, and more sophisticated player strategies will be implemented.

The tasks for this milestone are:

1. "Network Player" implementation
2. Automated player improvements
3. Cloud deployment for players
4. Code review

Timeline

This milestone is expected to be completed within roughly three one-week "sprints". For each sprint, you will plan work, submit your progress, and retrospect in order to identify ways to improve in the future.

Submission (each sprint)

These artifacts should be submitted at the end of each sprint.

1. (Team facilitator) Submit the sprint plan PDF on Moodle
2. (Team facilitator) Create a GitHub release of your repository and submit the zip file from your release on Moodle.
    a. Create a new tag for each release, named `sprint<#>`, replacing `<#>` with the sprint number. This milestone covers sprints 8-10.
3. (Team facilitator) Submit the end of sprint report PDF on Moodle
4. (All team members) Fill out the peer feedback form

When the milestone is complete, the team facilitator should create a GitHub release of your repository. Create a new tag for the release, named `milestone3`. The zip file of the release, along with the URL from story 3, should be submitted on Moodle. This additional submission step should be completed even if the code is the same as an end-of-sprint submission.

Updated 3/21/25

Grading

Grading for sprints uses the standard sprint grading criteria.

Grading for the milestone will focus on the implementation of your network player, the sophistication of your player improvements, the correctness of the HTTP server cloud deployment, and how well your team responded to code review feedback.

General Advice

The general advice from Milestone 1 still applies here. Some particularly important points for this milestone:

Be creative in sequencing and parallelizing your work. If you approach this work naively, there is a long sequence of work involved in implementing a network player, packaging it up into a Docker container, and deploying it to Azure. Completing these tasks serially may not be possible in the timeline for the milestone. It's critical to identify ways to make progress on multiple parts of this in parallel. For example, start by writing and checking in a very simple "hello world" HTTP webserver. Then, one team member can work on the Docker/Azure parts of the project using this simple webserver to test, while another works in parallel on the network player implementation. Combining these two work streams at the end likely won't be too complicated.

Don't underestimate the amount of time needed to adopt new technologies. For this milestone, you'll be writing a HTTP webserver, using Docker, and using Azure - likely all for the first time. Things will take longer than you expect. Plan with this in mind!

Updated 3/21/25

Story 1: Network Player

In this story, you will create the client and server for a network-based player.

The "player client" will implement the Player interface from the ATG API, and be used directly by the Engine. It will not contain any decision-making logic, and instead will request decisions from a remote HTTP server - the "player server". The player server will contain all the necessary decision-making logic, and respond with the chosen decision. The player client itself should not contain any player logic, and should be able to work with any player server which follows the server API specifications (found below).

The "player server" will be an HTTP server. It will use an "RPC" (remote procedure call) API style. It will accept POST requests with JSON bodies that represent the game state and possible decisions, and respond with a chosen decision.

Specifications

HTTP Server

The player server should expose two HTTP endpoints:

- `/decide (POST)`
  - Expects a JSON body formatted according to the "DecisionRequest" schema below
  - Successful (200) responses should include a JSON body formatted according to the "DecisionResponse" schema below.
  - Unsuccessful responses should use appropriate 4xx or 5xx HTTP response codes.
- `/log-event (POST)`
  - Expects a JSON body formatted according to the "LogEventRequest" schema below
  - Successful (200) responses should have empty bodies.
  - Unsuccessful responses should use appropriate 4xx or 5xx HTTP response codes.

Request / Response formats

The formats below are specified using the JSON Schema format.

DecisionRequest

```
{
  "title": "DecisionRequest",
```

```
  "description": "An object representing a request for a player to
choose a decision",
  "type": "object"
  "properties": {
    "state": {
      "description": "A serialized GameState object"
      "type": "GameState"
    },
    "options": {
      "description": "The list of possible Decisions"
      "type": "array"
      "items": {
        "description": "A serialized Decision object"
        "type": "Decision"
    },
    "reason": {
      "description": "A serialized Event object"
      "type": "Event"
    },
    "player_uuid": {
      "description": "A unique identifier for the player making the
request"
      "type": "string"
    }
}
```

DecisionResponse

```
{
  "title": "DecisionResponse",
  "description": "An object representing a decision chosen by the
player",
  "type": "object"
  "properties": {
    "decision": {
      "description": "A serialized Decision object"
      "type": "Decision"
    }
  }
}
```

LogEventRequest

```
{
  "title": "LogEventRequest",
```

```
  "description": "An object representing a game event that can be
logged",
  "type": "object"
  "properties": {
    "decision": {
      "description": "A serialized Event object"
      "type": "Event"
    },
    "player_uuid": {
      "description": "A unique identifier for the player this event
is being sent to"
      "type": "string"
    }

  }
}
```

In the schema above, several objects have "types" that correspond with types provided by the ATG API. These types are already configured to serialize themselves to JSON in the appropriate schemas, and so the details of those schemas are not included here.

Resources & Recommendations

Server language

Your HTTP server can be written in any language - this is one of the benefits of modularization at the system level. However, it is recommended to implement it in Java, so you can use your existing player implementations to make decisions, rather than needing to implement a new player from scratch. In Java, you also benefit from using all of the classes provided by the ATG API. But, if you'd prefer to try another language, you are free to do so.

Java Server Framework

There are many ways to build an HTTP server in Java. The recommended approach for this project is to use Spring. Spring is a very heavyweight, opinionated Java framework that makes it possible to implement a good, compliant webserver with very little code. The price of this, though, is that the Spring framework itself is large and complex. You only need to learn a tiny slice of it in order to get a webserver working, but identifying *which* tiny slice is a challenge.

Reference Repository

Here is a sample repo which contains a multi-module Maven project, with a Spring sub-module, and an example of installing the Guava module needed to support parsing/serializing Guava

objects to JSON: https://github.com/brandeis-cosi-103a/spring-server-example. This is a very close match to what you'll need to do to get a Spring server running!

Other resources

- How to write a RESTful service with Spring
    - Following this guide will get you an HTTP server that accepts GET requests and responds with JSON
- JSON parsing and Jackson
    - Jackson is the library which handles JSON serialization and parsing
    - This is what will be used to turn classes into JSON when making network requests, and to parse JSON back into classes when receiving a network request
    - If Jackson is on the classpath, Spring will use it automatically
    - The JSON serialization and parsing will almost seem like magic if you set things up correctly - if you find yourself doing any manual JSON parsing in the server, you are probably not on the right track.
    - Jackson tutorial (a general resource on Jackson)
- Guava
    - The ATG API uses Guava's immutable containers (e.g. ImmutableList), which Jackson can't handle by default. So, we need to customize the Jackson module by installing a Guava module in our Spring application.
    - Spring Boot: Customizing the Jackson ObjectMapper
    - Guava Spring module
    - StackOverflow relevant thread
    - Source code for JacksonAutoConfiguration
        - You don't need to look deeply at this file, but it is included so you can see that, even though the configuration seems to work "magically", that it is actually implemented through a sophisticated, well-encapsulated class.
- Spring Boot in Visual Studio Code
    - This VSCode extension can make it easier to create a Spring application in your project

Story 1 deliverables

1. Code for a player client: An implementation of the Player interface which makes decisions by making a network call to a remote server
2. Code for a player server: An HTTP webserver which can respond to the requests made by the player client.

Story 2: Automated Player improvements

In this story, you will either improve one of your existing automated players, or implement a new, more sophisticated automated player. A new automated player must be capable of purchasing and using at least 2 different action cards.

The definition of "more sophisticated" is up to you. Possibilities include:

- Players that incorporate the actions of other players into their strategy (e.g. buy Monitoring cards only if other players are buying attack cards)
- Strategies that adapt to the starting hands (e.g. drawing 5/2 cryptocoins vs. 3/4 cryptocoins on the first two hands)

A good litmus test is whether your new player performs better than your old player (according to your rating harness).

Story 2 deliverables

1. A new Player, or an improvement to an existing automated player
2. Unit tests for the new player
3. A JAR, generated by Maven, which includes only the player class(es).

Updated 3/21/25

Story 3: Cloud Deployment of Player Server

In this story, you will package your player server in a Docker container, and deploy it to Azure as an Azure Container App.

Story 3 deliverables

1. A Dockerfile which packages your network player code into a Docker image
2. A URL at which your network player is available

Resources & Tutorials

Docker

1. Docker extension for VSCode (optional, but recommended)
2. Spring Boot with Docker
3. Dockerfile reference

To build an image, use a "docker build …" command. If you are building on a laptop with Apple Silicon (an M1/2/3 chip), you will need to add "--platform linux/amd64" to your "docker build …" command.

To run a container based on the image, use a "docker run …" command

To test a running image, use curl to send requests to its HTTP endpoints. You can also try running your Engine with a NetworkPlayer, where the NetworkPlayer tries to connect to the server running in the Docker container. When testing everything locally, the decision endpoint address will be `"localhost:<port>/decide"`.

To debug a running container, run this command from your terminal to get a bash prompt inside the container: `docker run --rm -it -p 3000:3000/tcp <your image name>:latest /bin/bash`

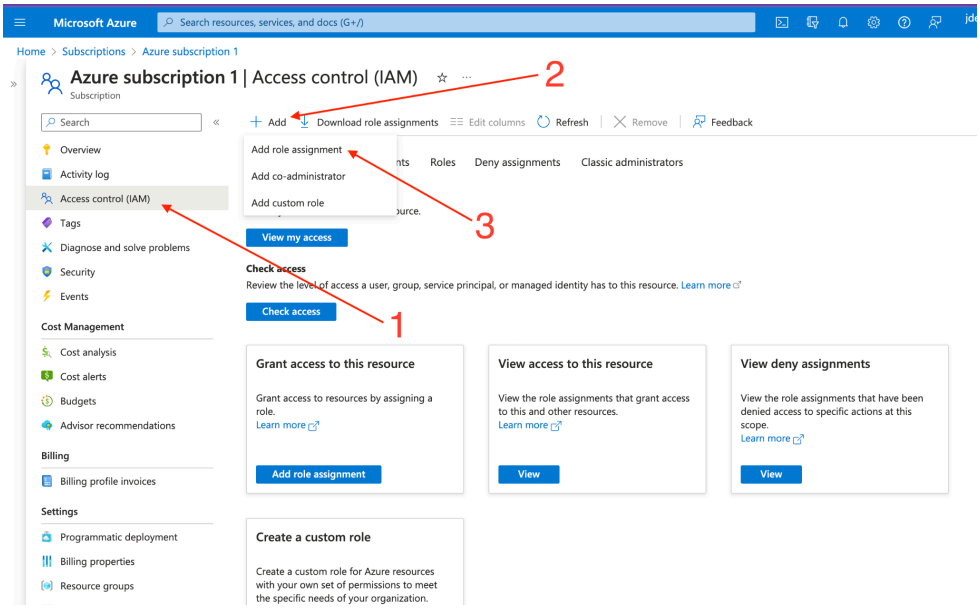
Azure account setup

Only one team member should do this part

1. Activate your Azure Student Account. Follow the link, click "Start free", and enter your Brandeis email address. You should not need to enter any payment information.
2. Navigate to "Subscriptions" in the Azure Portal, and select your subscription.

Updated 3/21/25

3. Click on "Access Control (IAM)" on the left, then "+ Add", then "Add role assignment" at the top:



4. Click on "Privileged Administrator Roles", then select "Contributor", then "Next" at the bottom.
5. Leave "User, group, or service principal" checked, then click "Select members".
6. Add each of your teammates, and each of the course staff by email.
7. Click "Review + assign" at the bottom.
8. Ask your teammates to try to log into the Azure Portal, to ensure they have access to your subscription.

Pushing a Docker image to the Azure Container Registry

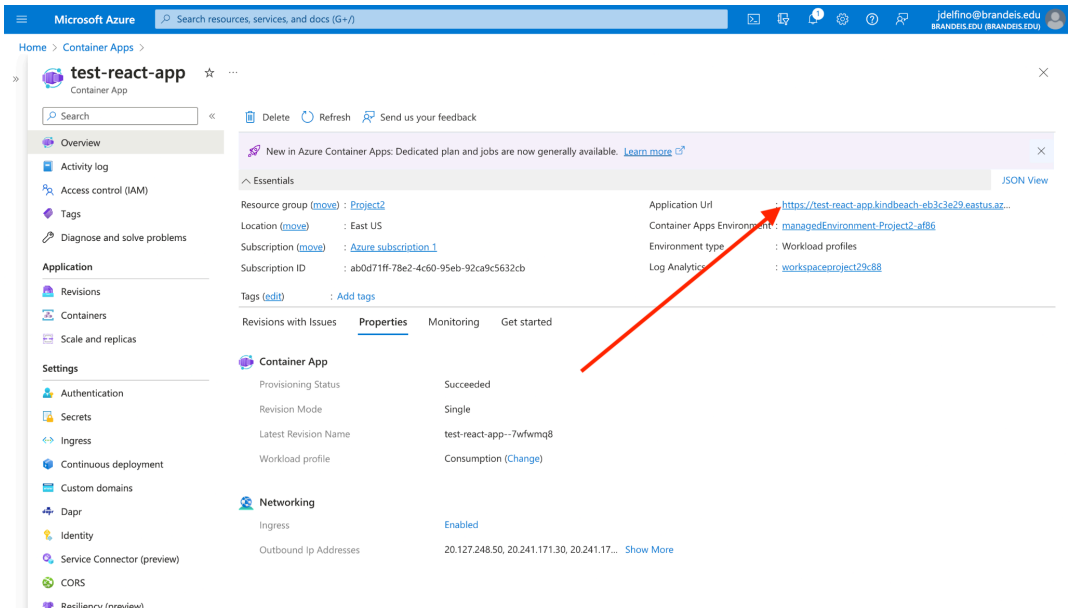
1. Create an Azure Container Registry
2. Enable an admin user on your registry (see https://stackoverflow.com/a/78834237)
3. Push your built image to the container registry using the "docker push …" command
    a. If you navigate to your new registry and choose the "push a container" button in the main section of the screen, it will display some sample commands and instructions.

Deploying a Docker image as an Azure Container App

1. Deploy a new Container App based on the image you just pushed
    a. Most options are either self-explanatory or can be left as-is, but here are a few tips:
    b. Don't use a "quickstart image" on the "Container" setup step
    c. Enable "Ingress" on the "Ingress" setup step
        i. Set it to "Accept traffic from anywhere"
        ii. Check "allow insecure connections"

iii. Enter the port your application listens on (probably 8080, unless you've changed defaults) in "Target port"

2. After deploying, click “Go to Resource”, then test it by sending a request to the "Application URL" for your app:



3. Test it for real by running your Engine, and using a Network Player pointed at the URL for your deployed HTTP server.

Updating an Azure Container App

Updating an Azure Container App is easy, but requires you to use explicit docker tags on your images, rather than reusing the default "latest" tag.

To update:

1. Rebuild your docker container, but add a tag (e.g. ":v1.1.0") to your image name. Use a new tag each time you make a change.
2. Push the new image to your container registry
3. Go to the "Application->Containers" menu in your Container App in the Azure Portal.
4. Modify the container to use your newly pushed tag, then click "Save as new revision"

Code review

During each major milestone, your group will need to schedule a code review with either the instructor or your team's assigned TA. The output of the code review is a combination of required and optional changes. Your team must implement the required changes in order to fully complete the milestone.

Timing of the code review

The best time for a code review is during the second sprint of a milestone, or early in the third sprint. This allows time for some changes from the milestone to be landed on the main branch, but also leaves enough time afterward to implement the required changes.

Preparation for the code review

Create a branch named `code-review-milestone-<#>` (replacing `<#>` with the number of the milestone) from the git tag that marked the release from your previous milestone.

Open a pull request from the tip of your main branch into this new branch. The description of the pull request should include a high level summary of the changes to be reviewed, and any other information that will help the reviewer navigate the changes.

Assign the course staff member who will be reviewing your code **and** the instructor as the "Reviewers" on the pull request. The code reviewer will leave their feedback on this pull request.

Code review process

Code reviews are performed synchronously - the reviewer and the team should be together (in person if at all possible, virtually if not) for the review. The entire team is strongly encouraged to attend; however, if schedule conflicts preclude this, a time should be chosen when at least 2 team members can attend.

The reviewer will "drive" (navigate the code on their laptop). The code review is intended to be collaborative and interactive - discussion between the reviewer and team is encouraged. The code reviewer will leave comments on the pull request during the review to summarize discussions and document suggestions.

Required changes will be explicitly labeled as such - any comment on a code review that is not clearly specified as required is optional. The code reviewer may take a short amount of time

after the review to mark the set of required changes, and will notify the team when the feedback is ready to be acted upon.