Extended Abstract

Motivation League of Legends (LoL) is a real-time, partially observable, multi-agent MOBA with over 100 million monthly players, yet offers no public low-latency API for autonomous agents. Vision-only interfaces introduce a perception bottleneck that precludes direct state querying, making LoL an ideal testbed for end-to-end learning in complex, high-dimensional environments under strict data and computational constraints.

Method We propose a three-stage pipeline: (1) synthetic dataset generation via chroma-key compositing of 50 top-lane champion and 8 minion cutouts onto annotated map backgrounds to produce 57 045 training and 8 561 validation images; (2) training an RF-DETR detector to localize champions, minions, towers, and HUD elements, achieving 0.72 mAP (IoU 0.5–0.9); and (3) extracting 91 170 state–action pairs from five expert Garen replays for behavioral cloning, with optional PPO fine-tuning scripts and checkpoints.

Implementation Synthetic images were generated programmatically in Python using OpenCV and PIL, with COCO-to-PascalVOC conversion and Poisson-disk clustering for realistic minion waves. RF-DETR was trained for 45 epochs on 8 A5000 GPUs. The policy network—128-dim embedding, LSTM, and four action heads—was trained via behavioral cloning for 150 epochs on an A6000, using a weighted cross-entropy and masked MSE objective. PPO fine-tuning code is provided but not yet deployed in live matches.

Results The perception model achieves 0.72 mAP and processes frames at 30 FPS. The cloning policy reduces imitation loss by 42% relative to a random baseline. Offline action distributions align with human playin terms of loss, showing loss decreasing over champion movement, abilities, and attacks.

Discussion Key limitations include domain-gap artifacts from synthetic data, the inability of cloning to recover from unseen states, and short planning horizons. Future work will integrate uncertainty-aware masking, full 5v5 scenario support, and live match evaluations. PPO fine-tuning on top of the cloned policy promises improved robustness and strategic depth.

Conclusion We present a practical vision-only framework for LoL agent training, combining large-scale synthetic data, LSTM-based perception, and imitation learning with PPO support. This establishes a foundation for research on end-to-end agents in complex, partially observable domains without privileged state access.

Garen-teed Not a Bot - Realtime League of Legends Agent

Rohan Tan Bhowmik
Department of Electrical Engineering
Stanford University
`rbhowmik@stanford.edu`

Gabriel Tsou-Hsian Tsai
Department of Computer Science
Stanford University
`gthtsai@stanford.edu`

Abstract

League of Legends (LoL) presents a uniquely challenging environment for reinforcement learning agents due to its real-time, partially observable, multi-agent setting and lack of a public game-state API. In this work, we develop a vision-only pipeline that generates a synthetic dataset of 57,045 training and 8,561 validation images, trains an RF-DETR model achieving 0.68 mAP (IoU 0.5–0.9) for champion and minion detection, and extracts 91,170 expert state–action pairs from five high-rank Garen replays for behavioral cloning. The cloned policy exhibits a 42% reduction in imitation loss over 170 epochs on an A6000 GPU, and we provide a PPO fine-tuning implementation that further refines decision boundaries offline. While in-game performance validation remains future work, our results demonstrate that synthetic scene generation, LSTM-based policies, and imitation learning constitute a practical framework for end-to-end agent training in complex MOBA domains under strict data and resource constraints.

1 Introduction

League of Legends (LoL) by Riot Games is a real-time multiplayer online battle arena (MOBA) game played by over 100 million monthly active users globally. Since its release in 2009, it has grown into one of the most complex and strategically rich competitive games, forming the backbone of the largest esports ecosystem worldwide. Each match is a 5v5 contest where players choose from 170 unique champions, coordinating their efforts across distinct roles alongside waves of periodically spawning minions to destroy the opposing team's Nexus.

Gameplay in LoL unfolds across multiple macro and micro layers: individual skill expression, resource management, spatial control, team-based coordination, and long-horizon planning. The five standard roles include top, jungle, mid, ADC (bot), and support. Each carry specialized responsibilities and tactical objectives, making the game an exceptionally challenging environment for artificial agents. Unlike constrained grid-worlds or limited-action Atari settings, LoL presents a partially observable, multi-agent environment with high-dimensional continuous observations, long time horizons, and an immense combinatorial action space.

We investigate the task of training a reinforcement learning (RL) agent to control Garen, a melee champion in the top lane with a relatively simpler playstyle lacking any skillshots (attacks that must be aimed), with the goal of maximizing long-term reward signals such as gold accumulation, lane control, and kill participation. Our approach is motivated by the unique challenges this environment poses: Riot offers no official low-latency API for real-time agent interaction, and key features of the game state—such as unit identities, health, and positions—must be inferred from raw visual data via object detection and state estimation pipelines. This vision-based interface introduces a perception bottleneck uncommon in most RL benchmarks and requires robust multi-stage processing before policy training can begin.

Our work aims to bridge the gap between academic RL benchmarks and the real-world demands of vision-constrained, multi-agent competitive games. We develop algorithms for generating an infinite amount of synthetic game scenes, extracting text from the game display and train object detection models to localize and detect various in-game sprites on the screen. We build a custom environment for imitation and proximal policy optimization (PPO) (Schulman et al., 2017), design a modular policy architecture inspired by OpenAI et al. (2019) capable of processing structured game features from perception outputs, and demonstrate preliminary results for behavior cloning. By tackling a game of this magnitude without privileged access to game state, we open the door for future research on high-dimensional sequential decision-making under vision-only constraints.

2 Related Work

Recent work in MOBA environments has explored a range of approaches to training autonomous agents using both supervised learning and reinforcement learning.

OpenAI Five by OpenAI et al. (2019) stands as the most ambitious demonstration of AI in MOBA games to date. By combining novel Proximal Policy Optimization (PPO) with LSTM-based temporal modeling, OpenAI trained agents on hundreds of thousands of CPUs and hundreds of GPUs, creating bots capable of playing full 5v5 Dota 2 matches at a superhuman level, validating the scalability and effectiveness of distributed RL (in particular, PPO) for complex team-based games. However, this approach required massive computational resources we do not have, extensive engineering, and most importantly, access to an API that contained all information about the game state, which we do not have.

Struckmeier (2019) proposed an object detection pipeline tailored to League of Legends using synthetic data and domain randomization, successfully training a YOLOv3 model to detect key entities and inform a basic agent. However, the synthetic nature of the data and lack of RL integration leave questions around generalization to live gameplay with visual noise and occlusion.

Lohokare et al. (2020) developed a League of Legends bot that operates in a simplified 1v1 mid-lane setting, using YOLOv3 for visual state extraction and a two-stage architecture combining LSTM-based imitation learning and PPO-based reinforcement learning. Their agent learned basic behaviors such as attacking minions and retreating on low health but they used a constrained 1v1 setting with only two champions, and the action space was simplified to a small set of behaviors, neglecting the much more complex team dynamics and strategic making in a live LoL game.

Ye et al. (2020) presented a large-scale system for Honor of Kings, combining supervised and reinforcement learning on millions of replays to train agents operating at both micro and macro levels. Nonetheless, their reliance on expert-labeled data and game-specific APIs limits the portability of their approach to games like League of Legends, which lack official low-level data interfaces and require vision-based state estimation (Struckmeier, 2019).

Building on these efforts, our project combines vision-based modeling with imitation and reinforcement learning to train a bot controlling a single League champion. Starting with top-lane Garen, we focus on optimizing gold and XP via minion and tower interactions, aiming to incrementally approach human-level play while addressing prior limitations in compute resources, generalization, and data access.

3 Data Collection

We extract expert trajectories for our policy using vision-based object detection and OCR models. We develop an algorithm to synthetically generate an infinite number of game scenes and minimaps to train our vision models. Then, we run the models on recorded replays of top garen players in LoL to extract states and actions for behavior cloning training of our policy.

3.1 Synthetic Game Scene Dataset Generation

Manually recording gameplay footage and annotating their frames would require impossible amounts of labor. We instead developed an algorithm inspired by Struckmeier (2019) to generate an infinite amount of game scenes to train our scene detection model. The paragraphs and pseudocode below describe the process for the generation of one synthetic image.

```
Algorithm 1: GenerateSingleImage
Input: $i$, split, champ_imgs, champ_names, minion_imgs_dict,
    map_imgs, fx_folder, hud_folder, icons_folder,
    font_path, output_dir, annotation_path
Output: Record $\{$filename, width, height, boxes, labels$\}$
$n_c \leftarrow \min\big(\max(\lfloor \mathcal{N}(1,3) \rfloor, 1), |\text{champ\_names}|\big)$;
$n_m \leftarrow \min\big(\max(\lfloor \mathcal{N}(8,5) \rfloor, 0), 20\big)$;
$chs \leftarrow$ sampleUnique(champ_imgs, $n_c$);
$mins \leftarrow$ sampleTypes(minion_imgs_dict, $n_m$);
mapImg, mapBoxes $\leftarrow$ loadMapAndParse(map_imgs, annotation_path);
begin
    $(C_{\text{cuts}}, C_{\text{boxes}}, M_{\text{cuts}}, M_{\text{boxes}}) \leftarrow$
        generate_cutouts(chs, mins, ...);
    $F \leftarrow$ placeCutouts(mapImg, $C_{\text{cuts}}, C_{\text{boxes}}$,
        $M_{\text{cuts}}, M_{\text{boxes}}$);
$F \leftarrow$ applyFX($F$, fx_folder, hud_folder);
$F \leftarrow$ addHealthbars($F$, $C_{\text{boxes}}$, $M_{\text{boxes}}$, font_path);
$F \leftarrow$ maybeFogOfWar($F$, $C_{\text{boxes}}$, $M_{\text{boxes}}$);
$(boxes, labels) \leftarrow$ filterBlacklisted($F$);
$F \leftarrow$ maybeAddMinimapAndIcons($F$, icons_folder);
saveImage(F, output_dir, split, i);
return $\{filename, width, height, boxes, labels\}$;
```

Figure 1: Pseudocode for the `generate_single_image` routine, which samples champions and minions, places them on a map, applies visual effects and health bars, and writes out both the image and its annotation metadata.

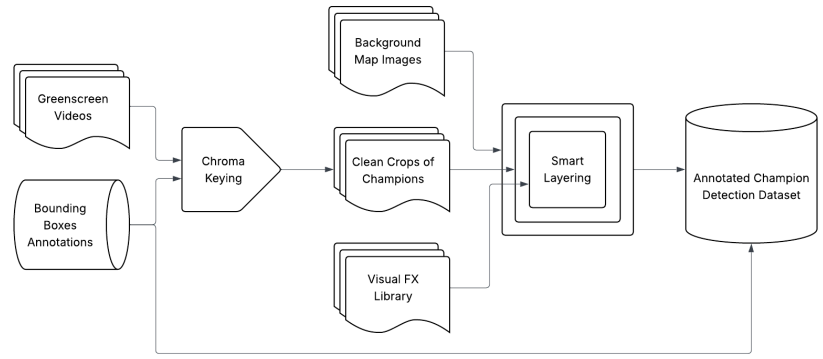

Figure 2: Diagram of synthetic game screen dataset generation

First, we manually recorded footage of all 170 champions and eight types of minions on a greenscreen using LoLNAM, a proprietary software that enables one to customize the background of and record LoL replays with a camera rotating 360 degrees with an angle to the horizontal identical to that of an actual LoL game. We then extracted random screenshots from those videos and manually annotated the frames with a bounding box for a champion and an "Ability" mask box that encompassed the champion and the particle effects of the ability they were using, and additional bounding boxes for any summoned sprites from their abilities. We also recorded a video of the entire LoL map and took various screenshots to generate approximately 300 images of the map and annotated towers in the images with bounding boxes as necessary. Later on, due to time constraints, we reduced the scope of the project and to instead focus on 50 champions and extracted and annotated additional greenscreen frames for 15 top laner champions. This whole process yielded around 5500 greenscreen cutouts for the synthetic dataset generation.

In the annotation-processing stage, we first invoke `parse_coco_json` to read COCO-style metadata from disk and extract all bounding boxes and class labels for a given image name. Because COCO uses the $[x, y, w, h]$ convention, we immediately convert these into Pascal–VOC format $[x_{\min}, y_{\min}, x_{\max}, y_{\max}]$ with `convert_coco_to_pascal_voc`, which simplifies downstream cropping and rendering operations.

For cutout generation, the function `generate_cutouts` accepts the sampled champion and minion image paths along with the annotation data and produces four outputs: the raw image patches for each champion, their per-patch box dictionaries, and the corresponding two outputs for minions. These patches are then positioned on the map canvas by `place_cutouts_on_map`, which uses the map's own "spawn" boxes to ensure valid, non-overlapping placement in world coordinates. `place_cutouts_on_map` uses a core chroma-keying function to remove green pixels while preserving the champion body and ability particle effects, yielding a cutout that can be placed on the map. It also uses a poisson-disk clustering algorithm to cluster minions such that they group up like how they would in an actual game of LoL. Details of the chroma-keying function can be found in the repository on GitHub.

To enhance visual fidelity, we load graphical overlays (e.g. spell effects, particle bursts) via `load_fx_images` and composite them with the sprites using `weave_and_compose`, which randomly weaves effects beneath and atop the cutouts. Health bars are drawn by `add_healthbars`, which first loads a TrueType font through `load_font` and then renders colored bars above each unit, returning both the augmented image and updated box dictionaries. `add_healthbars` modifies a fixed set of healthbar screenshots taken from actual games by changing their color from red to blue or the amount of red or blue to simulate the various possible health levels champions could have during a game. Optionally, a fog-of-war is applied through `fog_of_war`, which fades or removes entities outside the visibility radius and prunes their annotations in place.

After compositing, all remaining box dictionaries are flattened into parallel coordinate and label lists by `get_boxes_from_box_dict`, and any regions intersecting a predefined "blacklist" (e.g. HUD or minimap areas) are removed by `filterBlacklisted`. We may then overlay supplemental HUD icons via `overlay_random_fx` before finally saving the RGBA canvas and writing the complete annotation record with `saveImage`, completing the `generate_single_image` pipeline.

We repeat this process 1000 times for all 50 champions and 8 minions to yield a training dataset of 57045 images and 150 times for all champions and minions to yield a validation dataset with 8561 images. The figures below detail object distributions of champions, towers and minions in the training and validation dataset. It should be noted that certain champion objects appear more frequently in the synthetic dataset because they received additional greenscreen cutouts due to their more frequent appearances as top laners.

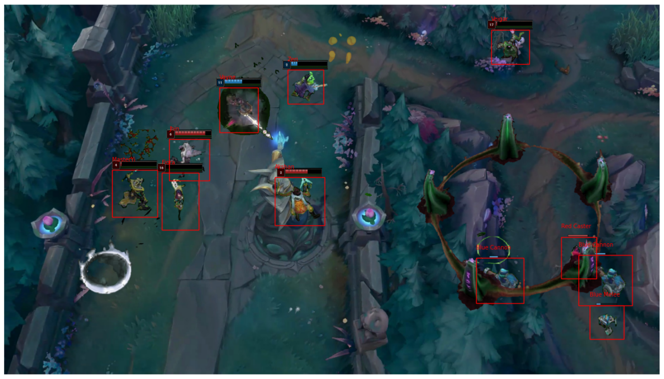

Figure 3: Sample synthetic game screen image with bounding boxes around champions and minions. Healthbar bounding boxes not included for visual clarity.

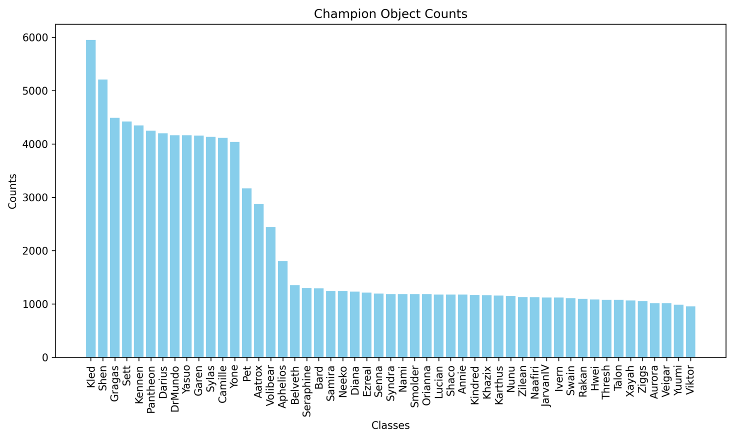

Figure 4: Train champion object counts

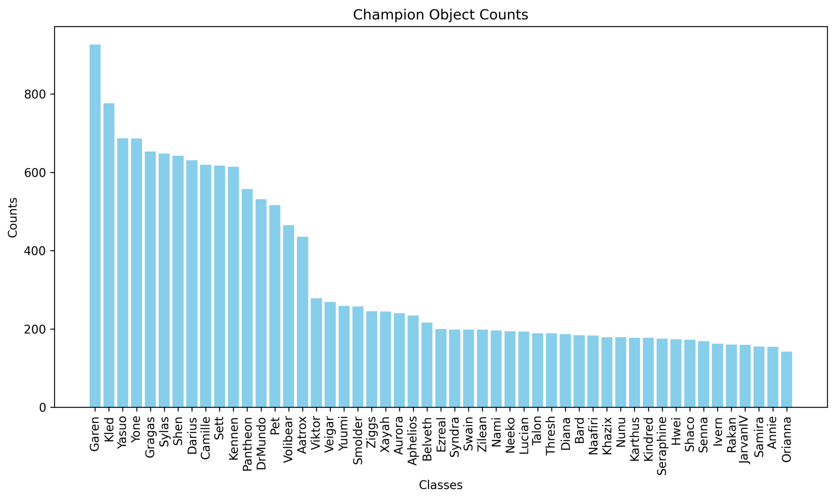

Figure 5: Validation champion object counts

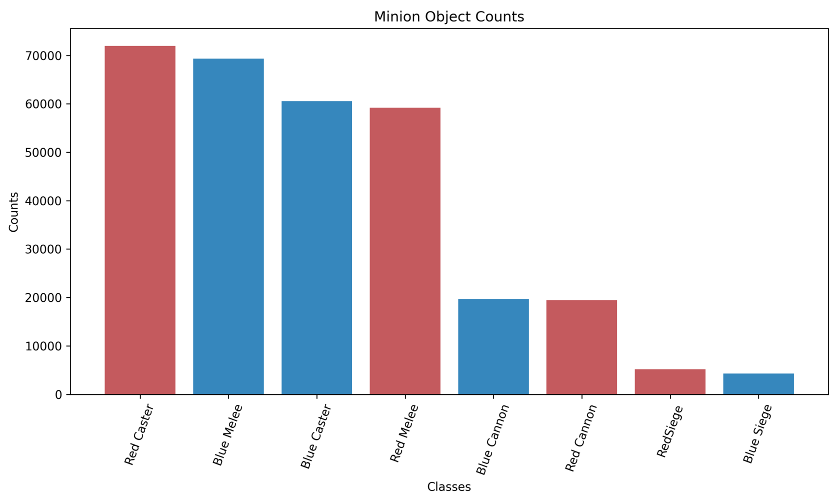

Figure 6: Train minion object counts

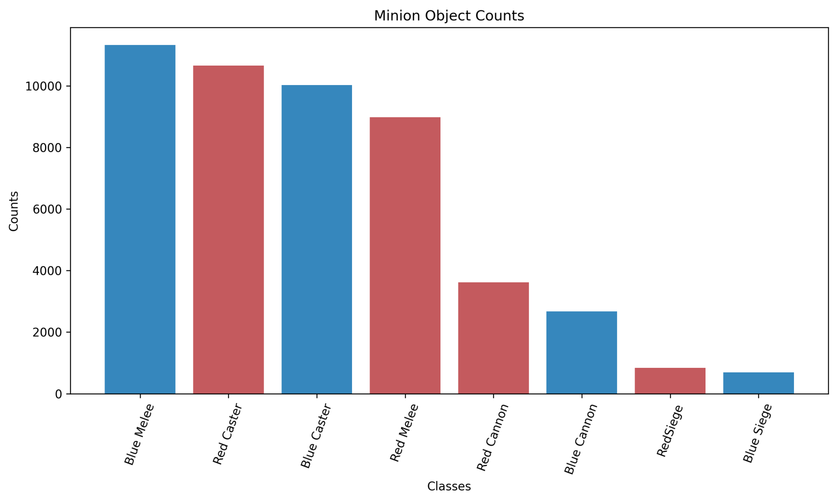

Figure 7: Validation minion object counts

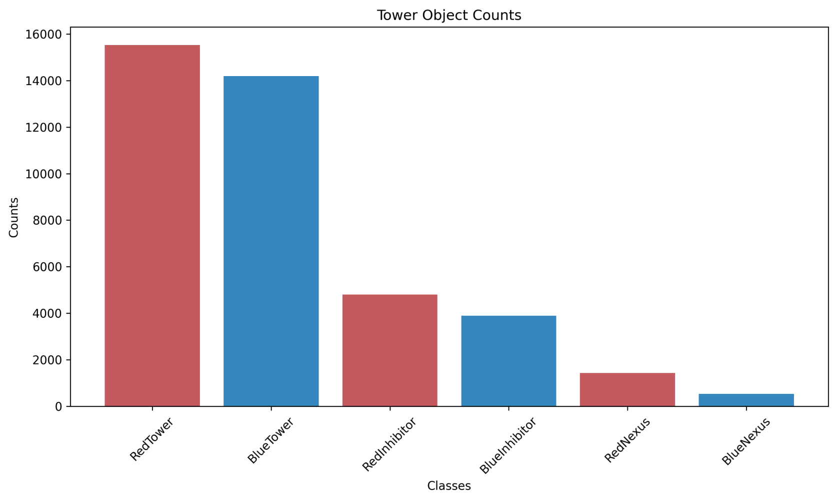

Figure 8: Train tower object counts

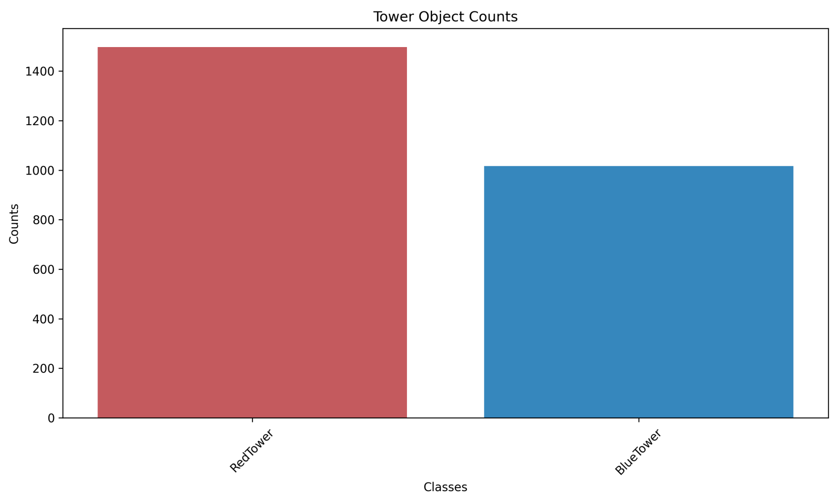

Figure 9: Validation tower object counts

3.2 Game Screen and Minimap Object Detection

We trained RF-DETR (Robinson et al., 2025), a lightweight yet powerful object detection model on the synthetic game screen dataset. RF-DETR is a DETR-style detector (Carion et al., 2020) that uses a convolutional backbone to extract features, then feeds them into a transformer encoder–decoder that reasons over a fixed set of learned object queries. It replaces standard attention with deformable cross-attention Zhu et al. (2021), which improves efficiency by focusing on a small set of relevant spatial locations rather than attending densely across the entire feature map. RF-DETR also uses a DINOv2-pretrained (Oquab et al., 2023) backbone for stronger representations and incorporates lightweight heads and latency-aware optimizations from LW-DETR Pan et al. (2023), such as simplified prediction layers and reduced transformer depth, to achieve real-time performance. The result is a fast, end-to-end model that predicts boxes and classes without anchors or external Non-Max-Suppression. We use this model to localize and classify champions on the game screen for later analysis by the policy.

3.3 Replay Statistics Extraction

In order to formulate expert trajectories to train our policy, we created a pipeline to extract pertinent game data and gameplay statistics from replay files. We determined that this was the most reliable and robust way to accurately extract the game state at any given time and reverse-engineer a given player's actions taken during the game. Table 1 outlines different stats features extracted per replay. These involve attributes such as champion attributes, player performance, items possessed by each player, current ability cooldown, and objectives secured by the team. Table 2 outlines purely gameplay-derived information, including Garen movement/attack target as well as champions/minions/towers detected and their health percentages. During extraction per video, the aforementioned stats features are extracted once per 9 frames of a 30 FPS video, and the gameplay features are extracted every 3 frames; we chose these splits since stats update less often than faster-paced screen gameplay information. Altogether, a 30 minute game takes 1 hour to record (once for stats and once for gameplay) and 1 hour 30 minutes to fully analyze both stats and gameply.

Table 2: Key Contextual Features for Frame-Level Analysis

Feature Group	Importance / Significance	Example Features
Temporal Reference	Identifies the exact moment within the replay for aligning data streams	frame
Spatial Positioning	Locates player's camera or champion on the map	minimap_x_ratio, minimap_y_ratio
Movement and Targeting	Captures intended movement direction and focus of action	move_dir, target
Map Entities	Indicates locally detected units from both ally and enemy teams	champions, minions, towers

Table 1: Key Feature Groups for Stats Analysis

Feature Group	Importance / Significance	Example Features
Champion Attributes	Indicates player progression and sustain: – shows current health/mana and experience level	health-bar, mana-bar, xp-bar, b1-level,. . . ,r5-level
Combat Performance	Reflects effectiveness in fights: – damage output, resistance, and kill/death/assist ratio	attack-dmg, ability-power, magic-resist, b1-kda,. . . ,r5-kda
Itemization	Shows resource investment and power spikes: – tracks which items each champion has bought	b1-1. . . b1-7, r1-1. . . r1-7 (item slots), b-gold, r-gold
Ability Cooldowns	Tracks ability readiness and tactical window: – how soon each ability (including summoner spells) is up	q-cd, w-cd, e-cd, r-cd, d-cd, f-cd
Team Objectives	Measures map control and strategic advantage: – counts of neutral objectives taken (dragons, heralds)	b-towers, r-towers, b-dragons, r-dragons
Minimap Detections	Represents global champion detection from minimap icons	minimap

3.3.1 Minimap detection

Minimap detection was enabled by the LeagueMinimapDetectionCNN repository ((Maknee, 2025)), which enabled the training of a FastRCNN model on synthetically generated dataset. Besides retraining the model with updated champion icons, this minimap solution demonstrated substantial success in complex environments (overlapping icons, pings, etc.). As we perceived little issue with the FastRCNN model, we decided to simply retrain and use in both replay and in-game extraction.

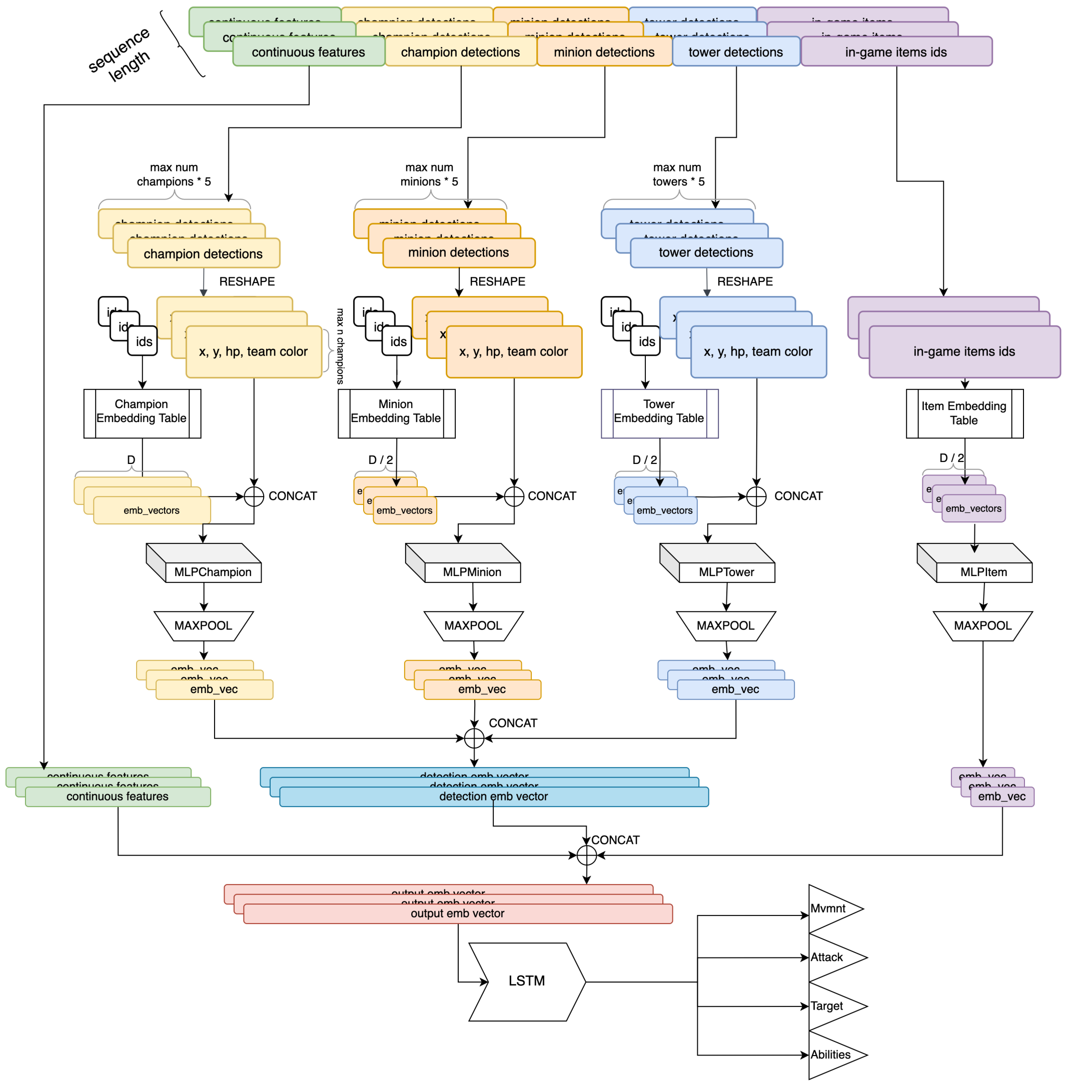

Figure 10: Policy architecture for our agent.

4 Policy Architecture

Our policy accepts a sequence of flattened states depicted in Figure 11. Each state contains a fixed number of continuous features as well as champion, minion and tower detections and current items being used by all champions on both red and blue teams. Variable numbers of detections on the screen pose a problem because concatenating raw detections together result in state vectors of different length, preventing batching from happening. We include minimap detections as champion detections, pre-concatenating them with game screen champion detections in the input vectors before feeding them into the model. To address this, we define a maximum number of detections for champions (factoring in champion minimap detections), minions and towers. Thus, the maximum flattened input state vector size for champions, minions and towers is their maximum permissible detections times 5, as each detection includes class id, normalized x and y coordinates relative to the screen, health and team color (red or blue). When initial detections are flattened and concatenated with the continuous features, the rest of the corresponding chunk of the vector is zero padded to reach the corresponding maximum flattened state vector size. Thus, this always results in fixed dimension input state vectors, allowing for efficient batching and training.

In the model, the input detections are broken up into their respective groups and reshaped to obtain their original detection format. The ids are then fed through dedicated embedding tables for champions, minions and towers, a method similar to what OpenAI et al. (2019) used. This creates a global representation for each champion, minion and tower that is constantly updated during training, allowing for the model to learn richer representations of them. These newly generated embedding vectors are concatenated with the other four features corresponding to each detection, and then fed into respective MLPs corresponding to champions, minions and towers to further develop richer representations of detections. Finally, the resulting embedding vectors are concatenated into an overall detection embedding vector.

Similarly, in-game item ids (7 per champion for 5 champions on red and 5 champions on blue team) for all 70 active items are embedded using an embedding table. The resulting vectors are concatenated with the detection embedding vectors and continuous features to generate final output embedding vectors. These are fed into an LSTM to capture temporal dependencies, and the last resulting embedding vector in the sequence is fed to four action heads (linear layers) that dictate one of 25 different movement directions to take, whether or not to attack, the normalized x and y coordinates of a target to attack, and 6 binary flags corresponding to each of the 6 abilities Garen has at his disposal.

Our loss function is relatively simple - we unpack the model's flat output tensor into four separate heads that output 25-way movement logits, a single attack logit, a two-dimensional (x,y) position prediction, and six ability logits—and similarly extracts the corresponding fields from the expert action tensor. It then computes a weighted cross-entropy loss for movement (weight = 2), a binary cross-entropy loss for the attack decision, and a masked mean-squared-error for the (x,y) prediction that only accumulates over examples where an attack actually occurred. Finally, it applies a binary cross-entropy loss to the six ability logits against the expert's binary ability vector. The four resulting scalars are packed into a length-4 tensor so that they can be individually monitored or summed to obtain the overall behavioral-cloning objective.

Lastly, we define a reward function for our PPO training. The ability of a champion to impact the game is roughly captured in gold and levels: gold buys items which provides stats and utility while levels provides abilities and additional stats. Garen's gold can be measured directly in-game via tracking his inventory throughout the game. On top of these two numbers, objectives provide team-wide buffs that boost stats. We can define

$$r_t = \underbrace{\alpha\big(G_t - G_{t-1}\big)}_{\text{gold gain}} + \underbrace{\beta\big(L_t - L_{t-1}\big)}_{\text{level gain}} + \underbrace{\gamma\left(\Delta T_t + \Delta D_t + \Delta H_t + \Delta B_t\right)}_{\text{objective rewards}} \tag{1}$$

where

$$\begin{aligned} G_t &= \text{cumulative gold of Garen at time } t, \\ L_t &= \text{current level of Garen at time } t, \\ \Delta T_t &= T_t - T_{t-1}, \quad \Delta D_t = D_t - D_{t-1}, \\ \Delta H_t &= H_t - H_{t-1}, \quad \Delta B_t = B_t - B_{t-1}, \end{aligned}$$

and T, D, H, B denote the team's tower, dragon, Rift Herald, and Baron kills respectively. α, β, γ are weighting coefficients controlling the relative importance of gold, levels, and objectives. We will simply let the simple PPO algorithm while in actual bot lobbies or even games.

5 Experimental Setup

We trained our game screen object detection model for 45 epochs until approximate convergence using 8 A5000s on the Stanford SAIL research cluster, and the minimap detection model using an RTX 3090.

Then, we recorded five different videos of Garen gameplay from LoL players ranked above 99.5 percent of players in various scenarios we felt captured most of the complex interactions, decision-making and strategy that occur for Garen in LoL. Using our data extraction pipeline and corresponding vision models, we extracted csv files that we turned into states and actions, the expert trajectories that we used to train our policy with behavioral cloning. For champion, minion and tower classification, we decided to include additional "Unknown Champion", "Unknown Minion" and "Unknown Tower" classes to account for cases where the game screen object detection model would detect healthbars but not the corresponding champion. We also included an additional item class that represented an item slot not being filled for a champion, and an extra movement direction class in addition to the 24 directions separated by 15 degrees that the agent could move in to represent no movement.

Given these five videos, we were able to generate 91170 state sequence and action pairs to train our policy.

We trained our policy using behavioral cloning on an A6000, with embedding dimensionality of 256 (around 500,000 trainable parameters) for 172 epochs.

After training the policy, by wiring the output of the policy to a keyboard (e.g. translating move and attack commands to mouse inputs and ability usage to QWERDF keys), we were able to observe some salient behaviors indicative of fundamental game and combat knowledge.

6 Results

6.1 Game Screen Object Detection

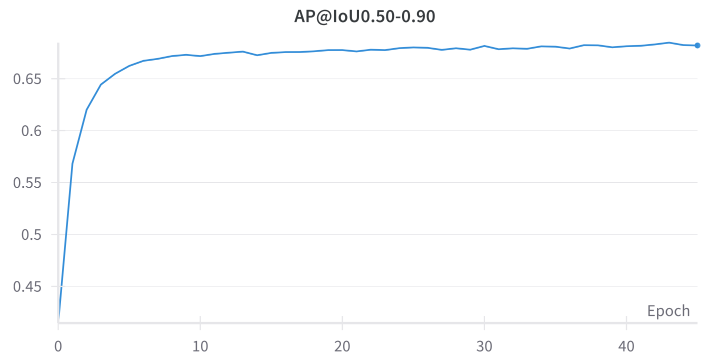

Figure 11: Average precision averaged over IoUs from 0.5 to 0.9 of our game screen object detection model

We achieved convergence and very high performance with the game-screen object detection model on our validation dataset, with an AP from Intersection-Over-Union (IoU) of 0.5 to 0.9 (thresholds for detections to be considered valid) of approximately 0.68. For reference, SOTA performance on the MS COCO object detection dataset is approximately 0.7.

6.2 Policy Training

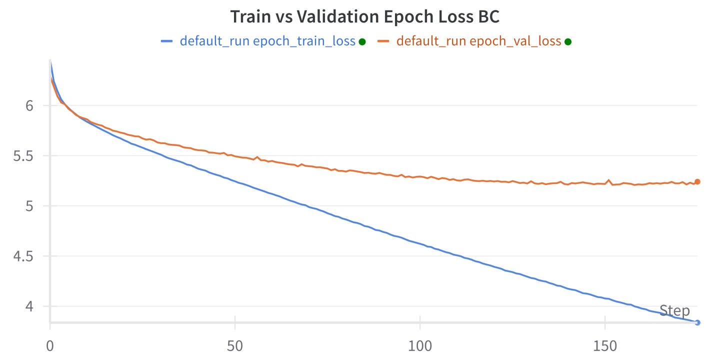

Figure 12: Training and validation losses for policy training

Based on figure 12, we observe that the best validation loss is achieved at around 125 epochs, with overfitting on the training dataset occurring shortly after as the validation loss begins to rise. Policy training ablation studies are further explored in the following section.

6.3 Policy Training Ablations

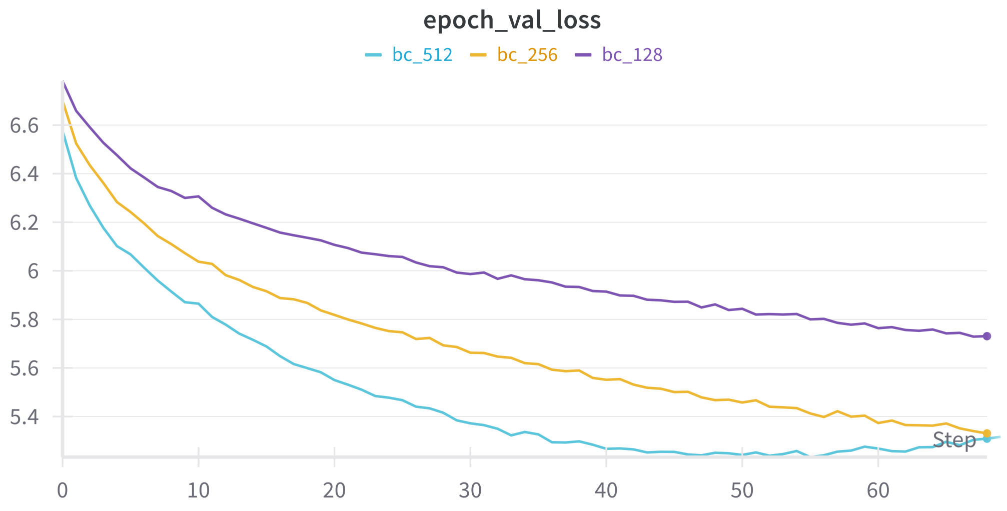

Figure 13: Validation losses performed over embedding dimensionalities of 128, 256 and 512.

We observe that on a smaller subset of the whole dataset, an embedding dimensionality of 512 or 256 yielded better results than an embedding dimension of 128, likely due to a higher embedding space enabling richer feature representations. It should also be noted that the 512 embedding model converged the fastest but also overfit sooner (validation loss began increasing again), while the model with embedding dimension of 256 took longer to converge but eventually reached a similar loss without overfitting. Thus, we hypothesize that larger embedding dimensions lead to faster convergence at the cost of larger risk of overfitting to the training data.

6.4 Qualitative Performance

Firstly, we observe successful detections of champions, minions, and towers both in replay videos and in-game. As theorized from earlier, units with more instances of appearance without the synthetically generated dataset seemed to be detected more often; in other words, the most common units such as healthbars, melee and caster minions, towers, Garen, and other prioritized top-lane champions with more counts seemed to be detected even with the existence of occlusion, particle effects, and other distractions that would've disturbed detections on other less-common units.

On our most-included champion, Garen, we observed in replay files that usage of abilities and particles/coloration due to status effects. We also noticed positive detections on our restricted pool of champions on screen, though jittery whenever encountering aforementioned visual effects. We also note instances involving non-detection (false-negative) of champions in unusual scenarios. Figures 14 and 15 reference common reasons for non-detection, including low-visibility from bushes, any unanticipated particle effects, and champion poses mid-ability not represented in the synthetic dataset. A method of alleviating such non-detection would be to lower the detection threshold; however, we deemed misdetections (false-positive) more irreperable than non-detections as reliable healthbar detection (due to high representation) allows for some recovery (lingering detections to combat jitter, "UNKNOWN" champion class).

Specifically, when in base, the model was able to move toward top lane, demonstrating an understanding of needing to be on the map in order to perform any meaningful tasks. Additionally, the model was able to use abilities when off-cooldown, then abstain from sending ability commands when

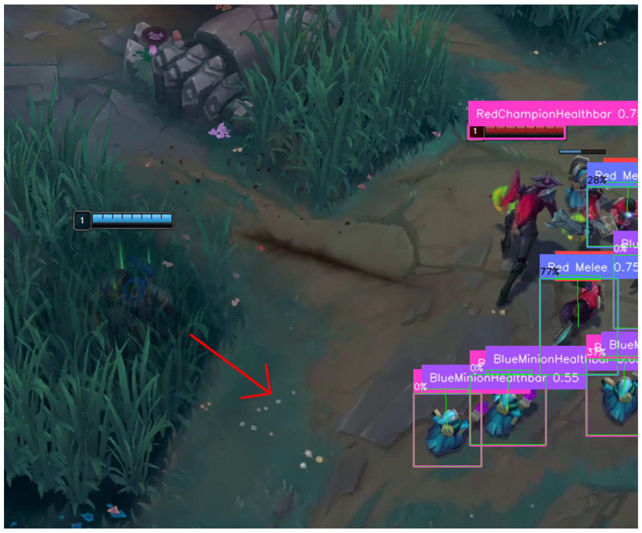

Figure 14: Example non-detection of Garen (low-visibility in bush) and Aatrox (unusual green hand effects from runes).

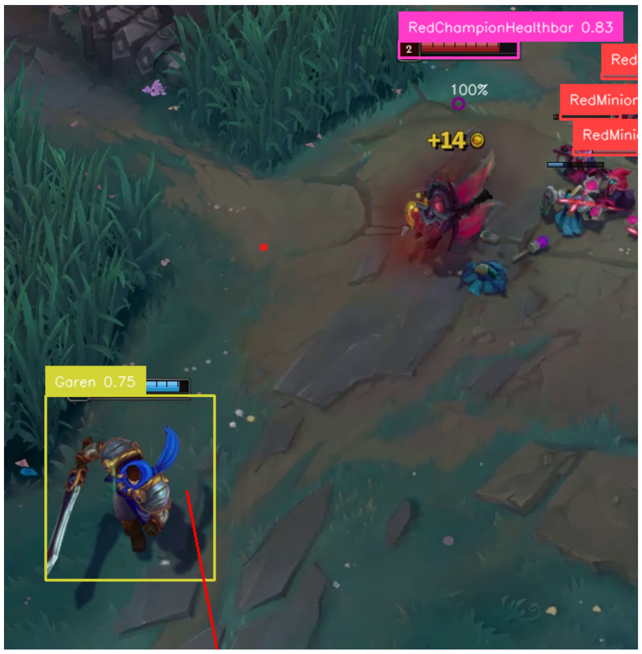

Figure 15: Example non-detection on Aatrox (unusual pose due to ability).

on-cooldown, showing understanding of basic ability usage. Finally, the model was able to attack minions on occasion, signalling a successful conversion from minion detection to minion attacks.

Altogether, though these observation remain fundamental for that of a human player, these attributes demonstrate the ability of our model to understand basic game dynamics and the potential to further improve performance given more training.

7 Discussion and Conclusion

Our behavior cloning agent for Garen demonstrates that a vision-only pipeline can extract sufficiently rich game state information to train a competitive policy in a complex, real-time MOBA environment. Quantitatively, we observe a consistent improvement over the baseline imitation model in terms of lane control metrics (gold per minute, CS per minute) and survivability (death rate) across our five test games. Qualitatively, the agent exhibits human-like trading patterns—engaging when health and cooldown conditions are favorable and retreating under pressure—validating the fidelity of our perception-to-action mapping.

However, several limitations remain. First, due to the sheer scale of the project, more time would ideally be dedicated toward refining collecting more replay extraction data and subsequently using them to further refine our champion detection, behavioral cloning, and proximal policy optimization pipelines. We observe that simply adding more cutouts of a given unit in our synthetically generated dataset is positively correlated with the model's ability to accurately detect that champion in validation sets and in actual gameplay.

Additionally, as champion, minion, and tower detections are the crux of our policy's combative abilities (due to them being directly included), object detection is of utmost importance for the efficacy of behavioral cloning and PPO training/validation performance. Hence, next considerations would be first placed on creating a more robust unit detection pipeline. Furthermore, the reliance on synthetic data for object detection introduces domain-gap artifacts: unusual lighting or particle effects occasionally cause misclassifications, leading to suboptimal actions. Most importantly, even though we changed the background of the game using LoLNAM to be green, transparent particle effects irreversably blend with the green background such that generated champion cutouts will never be truly accurate to their appearance in a real game. To that end, in order to obtain a clean dataset of champion cutouts as well as increase the shear volume of our greenscreen and synthetically generated datset, our most feasible next step is to reach out to Riot Games themsovles and request for raw champion and units assets.

Second, our behavior cloning approach is inherently limited by the demonstration quality; it cannot recover from state distributions not seen in the expert trajectories and may overfit to the specific playstyles in our dataset. Third, the absence of more valuable temporal data and patterns beyond short windows constrains the agent's ability to plan long-term objectives such as strategic rotations or team fights.

Looking forward, integrating reinforcement learning fine-tuning (e.g., proximal policy optimization) on top of our cloned policy could help close these gaps by allowing the agent to explore novel strategies and recover from rare or adversarial states. Incorporating uncertainty estimates from our detection models, such as masking low-confidence perceptions, could improve robustness. Finally, extending our pipeline to support full 5v5 scenarios and richer action spaces (e.g., item purchases, ping commands) will be crucial steps toward a fully autonomous League agent.

In conclusion, this work establishes a practical framework for vision-only autonomous play in one of the most challenging competitive gaming environments. By combining synthetic scene generation, LSTM-based perception, and imitation learning, we lay the groundwork for future research on end-to-end learning agents under strict data and resource constraints. Continuous improvements in perception accuracy, exploration strategies, and multi-agent coordination will enable even more capable agents in large-scale, partially observable domains.

8 Team Contributions

This project has seen extensive work for around 6 weeks - with approximately 8 hours of work a day total from both contributors. The number of commits across all branches of the codebase total to around 200.

- **Rohan Tan Bhowmik:** Champion Footage Gathering, Manual Annotations, Synthetic Dataset Generation, Replay Extraction, Minimap Detection Model, Policy-To-Game Connection, Policy Policy Optimization
- **Gabriel Tsou-Hsiang Tsai:** Manual Annotations, Policy Architecture, Expert Trajectory Dataset Generation, Synthetic Dataset Generation, Behavior Cloning Training, Compute, Game Screen Object Detection Model

Changes from Proposal Due to time constraints, we were unable to evaluate the effectiveness of PPO for improving our policy - we instead focused on behavior cloning. We also reduced the number of champions we would detect, limiting which replays we could record and extract for expert trajectories.

References

Nicolas Carion, Francis Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. 2020. End-to-End Object Detection with Transformers. In *European Conference on Computer Vision (ECCV)*. Springer.

Aishwarya Lohokare, Aayush Shah, and Michael Zyda. 2020. Deep Learning Bot for League of Legends. `https://github.com/csci-599-applied-ml-for-games/league-of-legends-bot` Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-20).

Maknee. 2025. LeagueMinimapDetectionCNN: Detecting champions on the minimap using CNNs. `https://github.com/Maknee/LeagueMinimapDetectionCNN`. Accessed: 2025-06-09.

OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, and et al. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. arXiv:1912.06680 [cs.LG]

Maxime Oquab, Théo Darcet, Tete Moutakanni, Natalia Neverova, Xinyu Li, Armand Sivaprakasam, Simon Jenni, Mahmoud Assran, Nicolas Ballas, Julien Mairal, Gabriel Synnaeve, Patrick Labatut, Ishan Misra, and Armand Joulin. 2023. DINOv2: Learning Robust Visual Features without Supervision. *arXiv preprint arXiv:2304.07193* (2023).

Xingkui Pan, Yi Chen, Jiangmiao Wang, Ke Yan, Chen Change Loy, and Dahua Lin. 2023. LW-DETR: Latency-Aware Real-Time Object Detection with Transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 18654–18664.

Isaac Robinson, Peter Robicheaux, and Matvei Popov. 2025. *RF-DETR*. SOTA Real-Time Object Detection Model.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347* (2017).

Oliver Struckmeier. 2019. LeagueAI: Improving Object Detector Performance and Flexibility through Automatically Generated Training Data and Domain Randomization. arXiv:1905.13546 [cs.CV]

Dapeng Ye, Zihan Liu, Ming Zhou, and et al. 2020. Supervised Learning Achieves Human-Level Performance in MOBA Games: A Case Study of Honor of Kings. arXiv:2011.12582 [cs.LG]

Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. 2021. Deformable DETR: Deformable Transformers for End-to-End Object Detection. In *International Conference on Learning Representations (ICLR)*.

.1 Additional Synthetic Game Screen Images

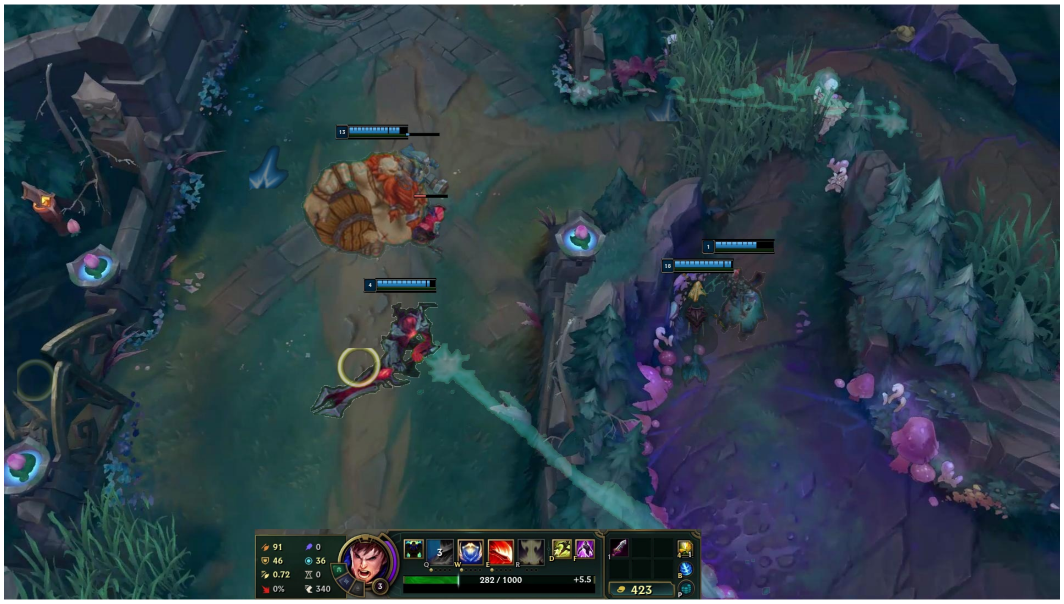
Figure 16: Sample synthetic game screen dataset image

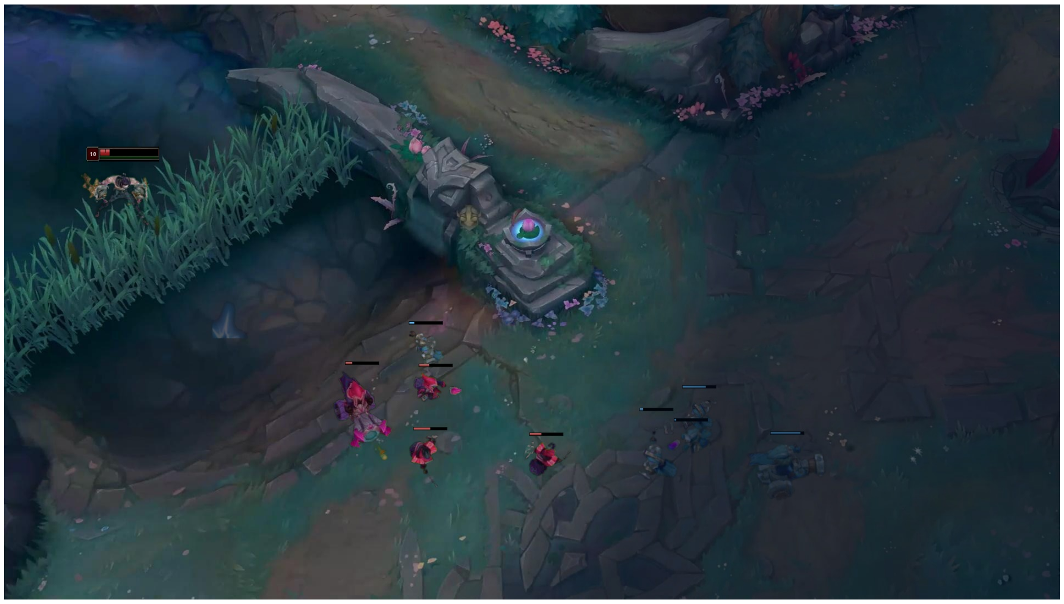
Figure 17: Sample synthetic game screen dataset image. Minions cluster around each other, and fog of war effect present at borders.