Projet CS:GO Esports Intelligence Platform

Pipeline MLOps et Stratégie de Monitoring

Équipe : Paul Roost, Axelle Desthombes, Alexis Bruneteau

Date : 2025-09-17

Dataset : CS:GO Professional Matches (Kaggle - 25K+ matches)
Objectif : Prédiction des résultats de matchs et optimisation des stratégies esports

1 Atelier 1 : Pipeline du Fil Rouge

1.1 Architecture Générale du Pipeline

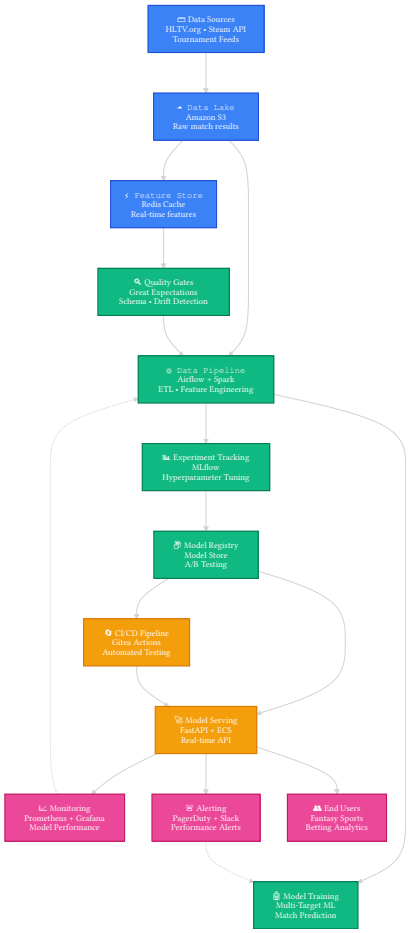

Figure 1: Architecture complète du pipeline MLOps CS:GO

1.2 Étapes Détaillées du Pipeline

1.2.1 Collecte et Ingestion des Données

Sources de données :

- **HLTV.org** : Résultats historiques, classements équipes

- **Steam API** : Données joueurs en temps réel
- **Tournament APIs** : Calendriers, formats de compétition

Pipeline d'ingestion automatisé avec Apache Airflow :

```
@dag(schedule_interval="@hourly", start_date=datetime(2024,1,1))
def csgo_data_ingestion():

    extract_hltv_matches = PythonOperator(
        task_id='extract_hltv',
        python_callable=scrape_hltv_matches
    )

    validate_data = PythonOperator(
        task_id='validate_raw_data',
        python_callable=validate_match_schema
    )

    store_s3 = PythonOperator(
        task_id='store_to_s3',
        python_callable=upload_to_s3
    )

    extract_hltv_matches >> validate_data >> store_s3
```

1.2.2 Feature Engineering Multi-Niveaux

Catégorie	Features
Team-level	• `recent_form_10_matches` - Ratio W/L récent • `map_pool_strength` - Win rate par map • `clutch_success_rate` - Performance clutch • `eco_round_conversion` - Gestion économique
Context	• `tournament_tier` - Prestige de l'événement • `prize_pool_amount` - Facteur de pression • `head_to_head_record` - Historique direct • `current_game_patch` - Version meta game
Live	• `current_score_difference` - Score en cours • `momentum_last_5_rounds` - Élan récent • `economy_advantage` - Avantage économique

1.2.3 Entraînement Multi-Target

Architecture d'apprentissage multitâche avec PyTorch :

```
class CSGOPredictor(nn.Module):
    def __init__(self, input_dim):
        super().__init__()
        self.shared_layers = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 128)
        )

        # Têtes spécialisées par tâche
        self.match_winner = nn.Linear(128, 2)    # Classification binaire
        self.final_score = nn.Linear(128, 2)     # Régression scores
        self.total_maps = nn.Linear(128, 4)      # Nombre de maps
```

```
    def forward(self, x):
        shared_repr = self.shared_layers(x)
        return {
            'match_winner': self.match_winner(shared_repr),
            'final_score': self.final_score(shared_repr),
            'total_maps': self.total_maps(shared_repr)
        }
```

1.3 Automatisation et Points de Contrôle

1.3.1 Stratégie d'Automatisation

Étape	Status	Justification
Ingestion données	AUTO	Nouveaux matchs quotidiens, obsolescence rapide
Feature Engineering	AUTO	Features dépendent de données temps-réel
Model Retraining	AUTO	Meta game évolue (patches, transferts)
Deployment	AUTO	Évite erreurs humaines, rollback rapide
Model Selection	MANUEL	Décisions business complexes nécessitant expertise

1.3.2 Points de Contrôle Critiques

Validation des Données :

```
def validate_match_data(df):
    """Validation avant feature engineering"""
    checks = [
        ('schema_compliance', validate_schema(df)),
        ('completeness', check_missing_values(df, threshold=0.05)),
        ('consistency', validate_team_names(df)),
        ('freshness', check_data_age(df, max_hours=24)),
        ('volume', validate_daily_match_count(df, min_matches=50))
    ]

    for check_name, result in checks:
        if not result.passed:
            raise DataValidationError(f"{check_name} failed")
```

Validation des Performances :

```
def validate_model_performance(model, validation_data):
    """Validation avant déploiement"""
    metrics = evaluate_model(model, validation_data)

    # Seuils minimaux
    assert metrics['accuracy'] > 0.65, "Accuracy insuffisante"
    assert metrics['roi_betting'] > 1.05, "ROI non profitable"
    assert metrics['upset_detection'] > 0.20, "Détection upsets faible"

    return True
```

1.3.3 Difficultés Techniques et Solutions

Défi 1 : Concept Drift Extrême

Les mises à jour du jeu modifient significativement les stratégies et l'équilibre, ce qui peut rendre les modèles existants moins performants.

Solution : Détection automatisée de drift + retraining d'urgence

```python
def detect_meta_shift(recent_matches, baseline):
    """Détecte changements post-patch"""
    map_rates = calculate_map_win_rates(recent_matches)
    baseline_rates = baseline['map_win_rates']

    for map_name in map_rates:
        ks_stat, p_value = ks_2samp(map_rates[map_name],
                                    baseline_rates[map_name])
        if p_value < 0.01:  # Drift significatif
            return True
    return False
```

Défi 2 : Cold Start Problem

Les nouvelles équipes ou changements de composition ne disposent pas d'historique suffisant pour l'entraînement.

Solution : Transfer learning via embeddings joueurs

```python
def handle_cold_start_team(roster, player_db):
    """Prédictions via similarité joueurs"""
    team_embedding = [player_db.get_embedding(p.id) for p in roster]
    similar_teams = find_similar_teams(team_embedding, top_k=5)
    return weighted_prediction_from_similar(similar_teams)
```

2 Atelier 2 : Expériences et Monitoring

2.1 Tracking des Expériences avec MLflow

2.1.1 Configuration et Logging Structuré

```
mlflow.set_tracking_uri("http://mlflow-server:5000")
mlflow.set_experiment("csgo-match-prediction")

def train_and_log_experiment(config):
    with mlflow.start_run(run_name=f"csgo-v{config.version}"):

        # Hyperparamètres
        mlflow.log_params({
            "model_type": config.model_type,
            "learning_rate": config.lr,
            "batch_size": config.batch_size,
            "data_version": config.data_version
        })

        # Métriques par époque
        for epoch in range(config.epochs):
            train_loss = train_one_epoch(model, train_loader)
            val_metrics = evaluate_model(model, val_loader)

            mlflow.log_metrics({
                "train_loss": train_loss,
                "val_accuracy": val_metrics['accuracy'],
                "betting_roi": val_metrics['roi'],
                "upset_detection": val_metrics['upset_rate']
            }, step=epoch)

        # Artefacts finaux
        mlflow.pytorch.log_model(model, "model")
        mlflow.log_artifacts("evaluation_plots/")
```

2.1.2 Métriques Trackées

Catégorie	Métriques
Performance ML	• Accuracy, Precision, Recall, F1-Score • ROC-AUC, Calibration Error • Performance par segment (tier tournoi)
Business	• ROI betting, Profit/Loss • Sharpe Ratio, Upset Detection Rate • User Engagement, Revenue Impact
Computational	• Training Time, Inference Latency • Model Size, Memory Usage • API Response Time

2.2 Stratégie de Monitoring Complète

2.2.1 Métriques de Surveillance Multi-Niveaux

Surveillance de la qualité des données :

```
class DataMonitoring:
    def monitor_data_quality(self, new_batch):
        metrics = {}
```

```
        # Volume et couverture
        metrics['daily_match_count'] = len(new_batch)
        metrics['team_coverage'] = new_batch['team_name'].nunique()

        # Qualité
        metrics['missing_rate'] = new_batch.isnull().mean().mean()
        metrics['duplicates'] = new_batch.duplicated().sum()

        # Drift distribution
        for col in ['team_ranking', 'match_duration']:
            drift = calculate_drift_score(new_batch[col], baseline[col])
            metrics[f'{col}_drift'] = drift

        return metrics
```

Model Performance Monitoring :

```
def monitor_model_performance(predictions, actuals):
    """Monitoring performance temps-réel"""
    rolling_metrics = {}

    # Fenêtres glissantes
    for window in [1, 7, 30]:  # jours
        recent = get_recent_data(window)
        rolling_metrics[f'accuracy_{window}d'] = accuracy_score(
            recent['actual'], recent['predicted']
        )
        rolling_metrics[f'roi_{window}d'] = calculate_roi(
            recent['predictions'], recent['outcomes']
        )

    return rolling_metrics
```

2.2.2 Système d'Alertes Intelligent

Sévérité	Seuils	Actions
CRITIQUE	• Accuracy 7j < 60% • ROI 7j < 100% • API errors > 5%	• PagerDuty + Slack • Email équipe oncall • Rollback automatique
WARNING	• Accuracy trending ↓ • Concept drift p<0.05 • Latency > 300ms	• Slack #alerts • Email ML team • Investigation requise
INFO	• Nouveaux tournaments • Performance updates • System health	• Slack #monitoring • Dashboard updates

2.2.3 Dashboards et Rapports

Dashboard Temps-Réel (Grafana) :

- **Model Performance** : Accuracy, ROI, Calibration trends
- **Data Pipeline Health** : Volume, freshness, quality scores
- **API Performance** : Latency P95, request rate, error rate
- **Business Metrics** : Revenue impact, user engagement

Rapports Hebdomadaires Automatisés :

```
class WeeklyReportGenerator:
    def generate_performance_report(self, week_start, week_end):
```

```
        sections = [
            self.executive_summary(),      # KPIs clés
            self.model_performance(),      # Analyse détaillée
            self.business_impact(),        # Valeur générée
            self.technical_health(),       # Infrastructure
            self.recommendations()         # Actions recommandées
        ]
        return self.compile_html_report(sections)
```

2.3 Architecture de Monitoring Production

2.3.1 Alerting Multi-Canal

```
class AlertManager:
    def __init__(self):
        self.channels = {
            'slack': SlackNotifier(SLACK_WEBHOOK),
            'email': EmailNotifier(EMAIL_CONFIG),
            'pagerduty': PagerDutyNotifier(PAGERDUTY_KEY)
        }

    def send_alert(self, alert):
        if alert['severity'] == 'CRITICAL':
            // Alertes critiques sur tous les canaux
            self.channels['pagerduty'].send(alert)
            self.channels['slack'].send_critical(alert)
            self.channels['email'].send_oncall(alert)
        elif alert['severity'] == 'WARNING':
            // Warnings vers Slack et email
            self.channels['slack'].send_warning(alert)
            self.channels['email'].send_team(alert)
```

2.3.2 Runbooks d'Incident

Alerte Critique : Accuracy < 60%

1. **Actions Immédiates (0-15min)**
   - Vérifier qualité des données récentes
   - Identifier changements meta/tournois
   - Rollback si accuracy < 55%
2. **Investigation (15-60min)**
   - Analyse drift sur données récentes
   - Comparaison prédictions vs résultats
   - Validation pipeline features
3. **Résolution (1-4h)**
   - Retraining d'urgence si drift détecté
   - Fix pipeline si problème data quality
   - Rollback si problème infrastructure

3 Conclusion

L'architecture MLOps développée pour ce projet CS:GO présente plusieurs caractéristiques importantes :

Architecture de production robuste :

- Apprentissage multi-tâches permettant des prédictions variées selon les besoins métier
- Service en temps réel respectant les contraintes de latence
- Gestion de la dérive conceptuelle liée à l'évolution du meta-jeu
- Surveillance complète des données, modèles et métriques business

Mesure de la valeur métier :
- Suivi du retour sur investissement pour les applications de paris et fantasy leagues
- Métriques d'engagement utilisateur pour optimiser la rétention
- Impact sur le chiffre d'affaires pour justifier les investissements

Fiabilité opérationnelle :
- Retour en arrière automatique en cas de dégradation des performances
- Système d'alertes multi-canaux pour une réaction rapide
- Procédures documentées pour la résolution d'incidents
- Plan de continuité d'activité pour les événements critiques

Ce travail démontre l'application des principes MLOps modernes à un domaine spécialisé, en mettant l'accent sur la création de valeur métier et la fiabilité opérationnelle.

Équipe MLOps - Projet CS:GO Intelligence Platform