LoadMagic.AI Documentation

Generated by: Philip Odzor

Date: February 25, 2026

Complete LoadMagic.AI documentation covering all topics from beginner to enterprise-level performance testing.

Contact: odzorp3@gmail.com

GitHub: https://github.com/odzorp

Table of Contents

Cover Page

LoadMagic.AI Enterprise Documentation System

> Complete Learning & Implementation Guide for Enterprise Performance Testing

---

Documentation Overview

This documentation system provides a comprehensive learning path from beginner to senior enterprise engineer, enabling you to build production-ready performance testing applications using LoadMagic.AI.

Learning Objectives

By completing this documentation system, you will be able to:

- Understand LoadMagic.AI's core capabilities and enterprise value
- Design and implement performance testing pipelines
- Integrate LoadMagic.AI into enterprise CI/CD workflows
- Build scalable, high-availability testing infrastructure
- Implement security, governance, and compliance controls

Prerequisites

Skill Level	Required Knowledge
Beginner	Basic understanding of APIs, HTTP requests, web applications
Intermediate	Experience with JMeter or Locust, CI/CD concepts
Advanced	Enterprise architecture, Kubernetes, Infrastructure as Code
Senior	Multi-region deployments, governance frameworks, compliance requirements

---

Quick Navigation

Section	Description	Target Audience
[1-Overview](./1-Overview/)	Core capabilities, enterprise value, use cases	All Levels
[2-Architecture](./2-Architecture/)	System architecture, components, terminology	All Levels
[3-Quickstart](./3-Quickstart/)	Setup, basics, Hello-World pipeline	Beginner
[4-Core-Usage](./4-Core-Usage/)	Data connectors, pipelines, monitoring	Intermediate
[5-Advanced-Engineering](./5-Advanced-Engineering/)	Optimization, scaling, HA patterns	Advanced
[6-Enterprise-Implementation](./6-Enterprise-Implementation/)	SSO, CI/CD, IaC, multi-tenant	Advanced/Senior
[7-Enterprise-Examples](./7-Enterprise-Examples/)	Real-world project implementations	All Levels
[8-Learning-Path](./8-Learning-Path/)	Progressive labs and exercises	All Levels
[9-Operations-Guide](./9-Operations-Guide/)	Operations, checklists, maintenance	Senior

---

Getting Started

For Beginners

Start your journey here:

1. [Overview](./1-Overview/) - Understand what LoadMagic.AI is and its value
2. [Architecture](./2-Architecture/) - Learn the system components and concepts
3. [Quickstart](./3-Quickstart/) - Set up your first project

For Intermediate Users

Deepen your knowledge:

4. [Core Usage](./4-Core-Usage/) - Master data connectors and pipeline orchestration
5. [Enterprise Examples](./7-Enterprise-Examples/) - Build real-world applications

For Advanced Engineers

Scale and optimize:

6. [Advanced Engineering](./5-Advanced-Engineering/) - Performance optimization and scaling
7. [Enterprise Implementation](./6-Enterprise-Implementation/) - Enterprise integration patterns

---

Tools & Technologies

This documentation covers integrations with:

Category	Technologies
Load Testing	Apache JMeter, Locust
Programming	Python, JavaScript/TypeScript, Groovy
CI/CD	Jenkins, GitLab CI, GitHub Actions, Azure DevOps
Cloud	AWS, Azure, GCP
Container	Docker, Kubernetes
IaC	Terraform, Ansible, CloudFormation
Monitoring	Prometheus, Grafana, Datadog

Support & Resources

- Official Website: https://loadmagic.ai
- Documentation: https://loadmagic.ai/more
- Book a Demo: https://loadmagic.ai/booking
- Contact:

License

Last Updated: February 2026

1. Overview

What is LoadMagic.AI?

Overview

LoadMagic.AI is an AI-powered performance testing platform that revolutionizes how organizations create, maintain, and execute load tests for their applications. By leveraging specialized AI agents, LoadMagic.AI automates the most tedious and error-prone aspects of performance testing with Apache JMeter and Locust frameworks.

Core Definition

LoadMagic.AI serves as an intelligent co-pilot for performance engineers, replacing manual scripting and correlation efforts with AI-driven automation that can generate production-ready test scripts in minutes rather than hours or days.

Key Distinction

Traditional Performance Testing	LoadMagic.AI Performance Testing

Manual script creation	AI-generated scripts from HAR files
Hours of correlation work	Automated correlation in minutes
Brittle scripts that break easily	Self-healing scripts that adapt to changes
Requires expert knowledge	Accessible to teams without deep JMeter/Locust expertise

The Problem LoadMagic.AI Solves

Performance testing has traditionally been one of the most time-consuming and technically demanding aspects of software quality assurance. The main pain points include:

1. Script Preparation Time
- Manually recording and scripting user flows can take 4-8 hours per script
- Complex authentication flows require expert knowledge
- Dynamic values (tokens, session IDs, timestamps) need careful correlation

2. Maintenance Overhead
- Every application change can break existing scripts
- Correlation rules must be updated whenever dynamic data patterns change
- Teams spend more time maintaining scripts than creating new tests

3. Knowledge Silos
- Only specialized performance engineers can create effective tests
- Organizational knowledge resides in individual expertise
- Onboarding new team members requires significant training investment

How LoadMagic.AI Works

The AI Agent Approach

LoadMagic.AI introduces five specialized AI agents that handle different aspects of performance testing:

LoadMagic.AI Platform

George Rupert Suzy
JMeter Genie Regex Expert Script
Scholar

Carrie Quinn
Correlator QA Assessor

Supported Frameworks

JMeter Locust

The Workflow

1. Record - Capture browser traffic as HAR file
2. Upload - Import HAR into LoadMagic.AI
3. Process - AI agents analyze and generate scripts
4. Execute - Run tests with automated correlation
5. Analyze - View results in Results Visualiser
6. Maintain - Self-healing pipeline fixes issues automatically

Supported Frameworks

Apache JMeter
- Desktop plugin for local development
- Full GUI for test plan creation
- Extensive plugin ecosystem support
- Enterprise-grade features

Locust (Python)
- Web-based AI Studio interface
- Zero-coding script generation
- Python script output for CI/CD integration
- Modern, developer-friendly experience

Target Users

User Type	How They Benefit
Performance Engineers	Faster script creation, automated correlation, self-healing tests
QA Engineers	Access to load testing without specialized expertise
Developers	Quick performance validation during development
Test Managers	Reduced time-to-market, better test coverage
DevOps Teams	Automated CI/CD integration, consistent quality gates

Summary

LoadMagic.AI is not just another testing toolit's an AI-powered transformation of the performance testing discipline. By automating the hardest parts of load testing script creation and maintenance, it enables organizations to:

- Move faster - Scripts that used to take days now take minutes
- Maintain less - Self-healing correlation reduces ongoing maintenance
- Scale teams - Enable more people to create performance tests
- Save money - Reduce consultant hours and accelerate delivery

---

Next: [Core Capabilities](./core-capabilities.md)

1.1 Core Capabilities

Core Capabilities

Overview

LoadMagic.AI provides a comprehensive suite of AI-powered capabilities that transform every aspect of performance testing. This document details each capability and its practical applications.

---

1. AI Agents

George - JMeter Genius

Purpose: Error debugging and performance testing knowledge

Capabilities:
- Answers any question about JMeter configuration and best practices
- Automatically scans for errors in real-time
- Identifies root causes and suggests precise fixes
- Provides step-by-step remediation instructions

Use Cases:
- Troubleshooting failed test executions
- Understanding JMeter configuration options
- Optimizing test plan performance
- Resolving connection and timeout issues

Performance Improvement:
- Traditional debugging: 30 minutes
- With George: 30 seconds
- Speed improvement: 60x faster

Rupert - Regex Researcher

Purpose: Automated regex and JSON path extraction

Capabilities:
- Auto-generates regex patterns for JMeter extractors
- Creates JSON path expressions for JSON responses
- Validates patterns before insertion
- Optimizes patterns for performance

Use Cases:
- Extracting dynamic values from responses
- Parsing complex JSON structures
- Capturing data from HTML responses
- Building reusable extraction patterns

Performance Improvement:
- Traditional manual pattern creation: 5 minutes
- With Rupert: 30 seconds
- Speed improvement: 10x faster

Suzy - Script Scholar

Purpose: Groovy scripting and code generation

Capabilities:
- Generates JSR223 scripts from plain text descriptions
- Converts scripts between languages (Java, Groovy, JavaScript)
- Automatically fixes script errors
- Creates custom solutions for complex scenarios

Use Cases:
- Data validation logic
- Dynamic value generation (timestamps, UUIDs)

- Complex conditional processing
- Error handling automation

Performance Improvement:
- Traditional script writing: 30 minutes
- With Suzy: 30 seconds
- Speed improvement: 60x faster

Carrie - The Correlator

Purpose: Automatic dynamic value correlation

Capabilities:
- Analyzes HTTP responses to identify dynamic values
- Recommends extraction strategies
- Orchestrates the full correlation pipeline
- Works seamlessly with other agents

Use Cases:
- Handling authentication tokens
- Correlating session IDs
- Managing view states
- Processing user-specific data

Performance Improvement:
- Traditional manual correlation: 8 hours
- With Carrie: 4 minutes
- Speed improvement: 120x faster

Quinn - QA Assessor

Purpose: Test plan quality analysis

Capabilities:
- Scans for missing assertions
- Identifies correlation flaws
- Detects dodgy configurations
- Provides automated fixes

Use Cases:
- Pre-deployment quality checks
- Test plan audits
- Best practices enforcement
- Configuration validation

Performance Improvement:
- Traditional manual review: 2 hours
- With Quinn: 10 seconds
- Speed improvement: 720x faster

---

2. HAR Processing

Supported Browsers

Browser	Status	Notes
Chrome	Supported	Full HAR export support
Firefox	Supported	Full HAR export support
Safari	Supported	Limited by browser
Edge	Supported	Full HAR export support

HAR Import Features

- Automatic endpoint filtering - Removes static resources
- Auth token extraction - Identifies and isolates authentication
- Transaction markers - Groups requests into logical flows
- Request/Response parsing - Full header and body analysis

---

3. Script Generation

JMeter Script Features

- Full test plan generation
- Automatic correlation insertion
- Proper extractor configuration
- Assertion generation
- Controller organization
- Variable management

Locust Script Features

- Pure Python output
- Proper request sequencing
- Header and body handling
- Task grouping
- Environment configuration
- CI/CD-ready output

---

4. Self-Healing Correlation Pipeline

The 5-phase self-healing pipeline automatically maintains script functionality:

Phase 1: Validation
- Boundary contamination checks

- Orphaned character detection
- Extractor sanity verification

Phase 2: Auto-Repair
- Scans for extraction failures
- Invokes Rupert for regex fixes
- Applies corrected patterns

Phase 3: Repeat Offender
- Classifies broken extractors
- Identifies suspicious patterns
- Auto-repairs recurring failures

Phase 4: Missed Candidates
- Scans for dynamic values missed in initial pass
- Auto-processes through agent mode
- Updates extraction rules

Phase 5: Smart Search
- Large value hints
- Prefix/suffix matching
- Hash mode for encrypted values

---

5. Results Visualization

Features

- Real-time metrics - Live execution monitoring
- JSON tree navigation - Visual JSON response explorer
- One-click extraction - Point-and-click data extraction
- Error highlighting - Visual error identification
- Performance graphs - Response time visualizations

- Transaction grouping - Business flow organization

Free Utilities (Included)

- JSON path extractor
- Auto Regex inserter
- Response visualization
- Basic reporting

---

6. Locust AI Studio

Capabilities

- Zero-coding HAR-to-Locust - Upload HAR, get Python script
- AI Correlation - Full self-healing pipeline
- Monaco Editor - VS Code-powered editing experience
- Visual Step Editor - Drag-and-drop test design
- Multi-file Projects - Complex test organization
- Script Versioning - Change tracking and rollback

Automation Modes

Mode	Description	Control Level
Passive	AI analyzes, you review	High
Suggest	AI proposes, you accept	Medium
Agent	Full automation	Low

---

7. Integration Capabilities

CI/CD Integration

- Jenkins plugin
- GitLab CI support
- GitHub Actions
- Azure DevOps
- Custom webhook support

Cloud Platforms

- Multi-cloud support (AWS, Azure, GCP)
- Multi-region deployment
- Data center options
- Hybrid cloud configurations

Enterprise Features

- SSO/SAML integration
- Role-based access control
- Audit logging
- API access
- Custom branding

Summary Table

Capability	Speed Improvement	Key Benefit
Script Generation	60x faster	Zero coding required
Correlation	120x faster	Self-healing
Error Debugging	60x faster	Instant fixes
Regex Creation	10x faster	Auto-validation
QA Assessment	720x faster	Automated audits

Next: [Enterprise Value](./enterprise-value.md)

1.2 Enterprise Value

Enterprise Value

Overview

LoadMagic.AI delivers significant business value for enterprise organizations by transforming performance testing from a bottleneck into a competitive advantage. This document quantifies the value proposition across multiple dimensions.

Financial Impact

Cost Reduction

Traditional Approach Costs

Activity	Hours Required	Hourly Cost	Total Cost/Script
Script Creation	4-8 hours	$100-200	$400-1,600
Correlation	4-8 hours	$100-200	$400-1,600
Debugging	2-4 hours	$100-200	$200-800
Maintenance/month	2-4 hours	$100-200	$200-800
Total per script	12-24 hours		$1,200-4,800

LoadMagic.AI Approach Costs

Activity	Hours Required	Hourly Cost	Total Cost/Script
Script Creation	5-10 min	$100-200	$8-33
Correlation	2-5 min	$100-200	$3-17

| Debugging | 1-2 min | $100-200 | $2-7 |

| Maintenance/month | 15-30 min | $100-200 | $25-100 |

| Total per script | 23-47 min | | $38-157 |

Annual Savings Example

Scenario: Organization with 50 active performance test scripts

Metric	Traditional	LoadMagic.AI	Savings
Annual script costs	$120,000-480,000	$3,800-15,700	$116,300-464,300
Maintenance costs	$12,000-48,000	$1,500-6,000	$10,500-42,000
Engineering time	600-1,200 hours	20-40 hours	580-1,160 hours

Productivity Gains

Time-to-Market Acceleration

Traditional Pipeline:

Requirements	Script Creation	Correlation	Testing	Release
2 weeks	1-2 weeks	2-3 days	1 week	1 day

Total: 4-6 weeks

LoadMagic.AI Pipeline:

Requirements	Script Creation	Correlation	Testing	Release
2 weeks	2-4 hours	30 min	1 week	1 day

Total: 3-4 weeks

Time saved per release cycle: 1-2 weeks

Engineering Productivity

Activity	Before LoadMagic.AI	After LoadMagic.AI	Reclaimed Time
Daily script work	2-4 hours	15-30 minutes	75%
Emergency fixes	4-8 hours	30-60 minutes	87.5%
Test plan reviews	2 hours	10 minutes	92%
Onboarding new engineers	2-4 weeks	2-4 days	85%

Strategic Value

Competitive Advantages

1. Faster Feature Delivery
- Reduced testing bottlenecks
- More frequent release cycles
- Earlier performance validation

2. Higher Quality
- Consistent test coverage
- Automated best practices
- Reduced human error

3. Scalable Testing
- Enable more teams to test
- Reduce specialist dependency
- Consistent quality gates

4. Risk Mitigation
- Catch performance issues earlier

- Automated regression testing
- Proactive monitoring

Organizational Impact

Team Transformation

Before LoadMagic.AI:

Specialists (Performance Engineers)

Manual Testing Bottleneck

Knowledge: Siloed
Capacity: Limited
Speed: Slow

After LoadMagic.AI:

All QA Engineers

AI-Assisted Testing

Knowledge: Democratized
Capacity: Scaled
Speed: Fast

Role Evolution

Traditional Role	Evolved Role	Value Add
Script Writer	Quality Architect	Strategy, design, analysis
Manual Tester	Test Automation Engineer	Framework development
Performance Specialist	Performance Strategist	Architecture, planning

---

Risk Reduction

Quality Metrics Improvement

Metric	Traditional	LoadMagic.AI	Improvement
Script defect rate	15-25%	2-5%	80% reduction
Correlation failures	30-40%	3-5%	87.5% reduction
Test execution time	100%	40-60%	50% faster
False positives	20-30%	5-10%	75% reduction

Compliance Benefits

- Audit trail logging
- Consistent test execution
- Reproducible results
- Documentation automation

ROI Calculation

Simple ROI Formula

ROI = (Annual Savings + Value of Reclaimed Time) - Annual Cost

---

Annual Cost

Example Calculation

Assumptions:
- 50 active scripts
- $150 average hourly rate
- 1,000 testing hours annually

Calculation:
- Annual savings: $290,000
- Value of reclaimed time: $150,000
- LoadMagic.AI annual cost: $25,000
- ROI = ($290,000 + $150,000 - $25,000) / $25,000 = 1,660%

Enterprise Readiness

Security & Compliance

Feature	Benefit
SSO/SAML	Enterprise authentication
RBAC	Fine-grained access control
Audit Logging	Compliance reporting
Data Encryption	Security at rest/transit
API Access	Automation integration
Custom Branding	White-label options

Support & SLAs

- 24/7 enterprise support

- Dedicated account management
- Custom SLAs
- Implementation assistance
- Training programs

Summary

LoadMagic.AI provides enterprise value across multiple dimensions:

Value Category	Key Benefit
Cost	90%+ reduction in scripting costs
Time	60-80% faster time-to-market
Quality	80%+ reduction in defects
Scalability	5x more testing capacity
Risk	Proactive issue identification
Compliance	Built-in audit and governance

Next: [Use Cases](./use-cases.md)

1.3 Use Cases

Use Cases

Overview

LoadMagic.AI supports a wide range of performance testing use cases across different industries and application types. This document details the most common and impactful use cases.

---

1. API Load Testing

Description

Test API endpoints under various load conditions to ensure they meet performance requirements.

Use Case Details

Target Applications:
- REST APIs
- GraphQL APIs
- gRPC services
- Microservices

Common Scenarios:

Scenario	Description	Key Metrics
Baseline Testing	Establish performance benchmarks	Response time, throughput
Peak Load	Test at expected peak traffic	Max users, error rate
Stress Testing	Find breaking points	Capacity limits
Spike Testing	Handle sudden traffic surges	Recovery time
Endurance Testing	Sustained load over time	Memory leaks, degradation

Example Workflow

python
 Locust script generated by LoadMagic.AI
from locust import HttpUser, task, between

class APIUser(HttpUser):
    wait_time = between(1, 3)

    @task(3)

```
    def get_users(self):
        self.client.get("/api/users")

    @task(2)
    def get_user_detail(self):
        user_id = self.environment.runner.user_classes[0].user_id
        self.client.get(f"/api/users/{user_id}")

    @task(1)
    def create_user(self):
        self.client.post("/api/users", json={
            "name": "Test User",
            "email": "test@example.com"
        })
```

---

2. Web Application Testing

Description

Simulate realistic user interactions with web applications to test overall system performance.

Use Case Details

Target Applications:
- E-commerce platforms
- Content management systems
- Customer portals
- Internal business applications

Key Features:
- Browser-based user flow simulation
- Multi-page transaction testing

- Session management
- Form submissions
- Shopping cart operations

Example Workflow

1. Record user flow in browser
2. Export HAR file
3. Upload to LoadMagic.AI
4. Generate correlated script
5. Configure load parameters
6. Execute and analyze

---

3. Microservices Performance Testing

Description

Test individual services and their interactions in a microservices architecture.

Use Case Details

Challenges Addressed:
- Service-to-service communication
- Distributed tracing requirements
- Database query performance
- API gateway overhead
- Load balancing effectiveness

Testing Strategy:

Microservices Architecture

API Service DB
Gateway A Cluster

Service
B

Service Cache
C Redis

Test Types

Test Type	Purpose	Focus
Service Load	Individual service capacity	Response time per service
Integration	Service communication	Latency, error rates
Chaos	Failure scenarios	Resilience, recovery
Contract	API compatibility	Version compatibility

---

4. CI/CD Integration

Description

Integrate performance testing into continuous integration/deployment pipelines.

Use Case Details

Benefits:

- Catch performance regressions early
- Automated quality gates
- Consistent testing across environments
- Faster feedback loops

Integration Points:

yaml
 GitHub Actions Example
name: Performance Tests

on:
  pull_request:
    branches: [main]

jobs:
  performance-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Generate JMeter Script
        run: |
          Upload HAR and generate script
          curl -X POST https://api.loadmagic.ai/generate \
            -H "Authorization: Bearer ${{ secrets.LOADMAGIC_TOKEN }}" \
            -d @har-files/test-scenario.har \
            -o jmeter-scripts/test.jmx

      - name: Run Performance Test
        run: |
          docker run -v $(pwd):/tests \
            jmeter:latest -n \

```
          -t /tests/jmeter-scripts/test.jmx \
          -l results.jtl

      - name: Analyze Results
        run: |
          python scripts/analyze_results.py results.jtl

      - name: Publish Results
        uses: actions/upload-artifact@v3
        with:
          name: performance-results
          path: results.jtl
```

---

5. E-commerce Performance Testing

Description

Test high-traffic e-commerce platforms during peak shopping periods.

Use Case Details

Critical Scenarios:

Scenario	Traffic Pattern	Duration
Black Friday	100x normal	24-48 hours
Flash Sales	50x normal spike	1-2 hours
Holiday Shopping	5-10x normal	Weeks
Product Launches	20x normal	Hours

Key Transaction Flows:

1. Browse - Product search, category navigation
2. View - Product detail, reviews
3. Cart - Add to cart, update quantities
4. Checkout - Address, payment, confirmation
5. Post-purchase - Order status, returns

Performance Thresholds

Metric	Target	Critical
Page load time	< 2 seconds	> 5 seconds
Search response	< 500ms	> 2 seconds
Add to cart	< 1 second	> 3 seconds
Checkout	< 10 seconds	> 30 seconds
Error rate	< 1%	> 5%

---

6. Financial Services Testing

Description

Test transaction processing systems in banking and financial applications.

Use Case Details

Compliance Requirements:
- Audit trails
- Data isolation
- Transaction integrity
- SLA compliance

Test Scenarios:

python

```
 Transaction processing test
class FinancialUser(HttpUser):

    @task
    def account_balance(self):
         Test account balance retrieval
        self.client.get("/api/accounts/balance")

    @task
    def fund_transfer(self):
         Test money transfer
        self.client.post("/api/transfers", json={
            "from_account": "ACC001",
            "to_account": "ACC002",
            "amount": 1000.00,
            "currency": "USD"
        })

    @task
    def transaction_history(self):
         Test transaction queries
        self.client.get("/api/transactions?limit=50")
```

---

7. Healthcare Application Testing

Description

Test patient-facing and provider-facing healthcare applications.

Use Case Details

Key Requirements:

- HIPAA compliance
- Data privacy
- High availability
- Quick response times

Test Scenarios:

- Patient portal access
- Appointment scheduling
- Medical record retrieval
- Prescription management
- Telemedicine sessions

---

8. Gaming Platform Testing

Description

Test multiplayer gaming platforms and real-time interactions.

Use Case Details

Challenges:

- Real-time latency requirements
- Concurrent user connections
- WebSocket handling
- State synchronization

Test Types:

- Login/authentication storms
- Matchmaking performance
- Real-time updates
- Leaderboard operations

- Chat/inventory operations

---

9. IoT Platform Testing

Description

Test IoT platforms that manage device connections and data processing.

Use Case Details

Scale Considerations:
- Millions of devices
- High throughput data ingestion
- Real-time processing
- Device command/response

Test Scenarios:
- Device registration bursts
- Telemetry data ingestion
- Command delivery
- Firmware updates
- Alert notifications

---

10. Mobile App Backend Testing

Description

Test backend services that power mobile applications.

Use Case Details

Considerations:

- Mobile network conditions
- Offline/online sync
- Push notification delivery
- API versioning

Test Scenarios:
- App launch authentication
- Content synchronization
- Image/media uploads
- Push notification delivery
- Background data sync

Summary Table

Use Case	Industry	Primary Focus	Complexity
API Load Testing	All	Throughput, latency	Low
Web Application	Retail, Media	User experience	Medium
Microservices	Tech, Finance	Service communication	High
CI/CD Integration	All	Automation	Medium
E-commerce	Retail	Peak handling	High
Financial Services	Finance	Transaction integrity	High
Healthcare	Healthcare	Compliance, availability	High
Gaming	Entertainment	Real-time performance	High
IoT Platforms	Manufacturing	Scale, throughput	Very High
Mobile Backend	Tech, Retail	API performance	Medium

Next: [Architecture Overview](../2-Architecture/architecture-overview.md)

2. Architecture Overview

Architecture Overview

Overview

LoadMagic.AI employs a modern, cloud-native architecture designed for scalability, reliability, and enterprise-grade security. This document provides a comprehensive overview of the system architecture.

High-Level Architecture

LoadMagic.AI Platform

Client Layer

Web UI (Studio) JMeter Plugin REST API

API Gateway Layer

Auth & Security Rate Limiting Request Routing

Core Services Layer

HAR Script Correlation
Processing Generation Engine

AI Results User
Agents Visualizer Management

Data & Storage Layer

Document AI Results
Store Model Store
(MongoDB) Storage (S3/GCS)

Client Layer

1. Web UI (Locust AI Studio)

The browser-based interface for script creation and management.

Components:
- Monaco code editor (VS Code powered)
- Visual step editor
- Project file manager
- Results dashboard

Technology Stack:
- React + TypeScript
- Monaco Editor
- WebSocket for real-time updates

2. JMeter Plugin

Desktop application for local development.

Components:
- JMeter integration
- Local script management
- Direct execution
- Results visualization

Requirements:
- Java 11+
- Apache JMeter 5.5+
- 4GB RAM minimum

3. REST API

Programmatic access for automation.

Endpoints:
- /api/v1/projects - Project management
- /api/v1/scripts - Script CRUD
- /api/v1/executions - Test execution

- /api/v1/results - Results retrieval
- /api/v1/agents - AI agent interaction

API Gateway Layer

Authentication & Security

Supported Methods:
- API Keys
- OAuth 2.0 / OIDC
- SAML 2.0 (Enterprise)
- JWT tokens

Security Features:
- TLS 1.3 encryption
- Request validation
- CSRF protection
- Input sanitization

Rate Limiting

Default Limits:

Plan	Requests/min	Concurrent Executions
Free	60	1
Pro	300	5
Enterprise	Unlimited	Unlimited

Core Services Layer

1. HAR Processing Service

Responsibilities:

- HAR file ingestion
- Request/response parsing
- Endpoint extraction
- Static resource filtering

Data Flow:

HAR File Upload Parse & Validate Extract Endpoints Filter Static Store Processed Data

2. Script Generation Service

Responsibilities:

- Template-based generation
- AI-powered scripting
- Code optimization
- Syntax validation

Supported Output Formats:

- JMeter (.jmx)
- Locust (.py)
- k6 (.js)

3. Correlation Engine

Responsibilities:

- Dynamic value detection
- Extractor generation
- Correlation validation
- Self-healing orchestration

Correlation Pipeline:

1. Response analysis
2. Candidate identification

3. Extractor creation
4. Validation testing
5. Production deployment

4. AI Agents Service

Agent Architecture:

AI Agents Infrastructure

LLM Integration Layer

Claude (Primary) GPT-4 (Secondary)

5. Results Visualizer

Features:
- Real-time execution monitoring
- Response time graphs
- Error categorization
- Transaction breakdowns
- Export capabilities

Data Layer

Document Store (MongoDB)

Collections:
- Projects
- Scripts
- Execution configs
- User preferences
- Audit logs

Object Storage (S3/GCS)

Stored Data:
- HAR files
- Generated scripts
- Test results (JTL)
- Execution logs

- Report exports

AI Model Storage

Models:
- Fine-tuned agent models
- Embedding caches
- Feature extraction models

Deployment Options

Cloud (SaaS)

Regions:
- US East (Virginia)
- US West (Oregon)
- EU (Frankfurt)
- UK (London)
- APAC (Singapore)

Benefits:
- Zero maintenance
- Automatic scaling
- Latest features

Private Cloud

Options:
- AWS PrivateLink
- Azure Private Endpoint
- Google Cloud Private Service Connect

Benefits:

- Data residency
- Custom networking
- Compliance control

On-Premises

Requirements:
- Kubernetes cluster
- 16+ CPU cores
- 64GB+ RAM
- 500GB+ storage

Security Architecture

Security Layers

1. Network Security
   WAF
   DDoS protection
   VPC isolation

2. Application Security
   Input validation
   Output encoding
   SQL injection prevention

3. Data Security
   Encryption at rest (AES-256)
   Encryption in transit (TLS 1.3)
   Key management (HSM)

4. Access Control
   RBAC
   SSO/SAML
   API key management

Scalability Design

Horizontal Scaling

Stateless Services:
- API Gateway
- Script Generation
- HAR Processing
- Results Visualizer

Auto-scaling Triggers:
- CPU utilization > 70%
- Request queue depth > 100
- Memory utilization > 80%

Vertical Scaling

Stateful Services:
- Database (MongoDB)

- Cache (Redis)
- AI Agent Queue

Summary

LoadMagic.AI architecture provides:

Layer	Components	Key Features
Client	Web, JMeter, API	Multiple interfaces
Gateway	Auth, Routing	Enterprise security
Core	Processing, AI	Intelligent automation
Data	Storage, Models	Secure, scalable

Next: [Components & Data Flow](./components-data-flow.md)

2.1 Components & Data Flow

Components & Data Flow

Overview

This document details the key components of LoadMagic.AI and how data flows through the system during typical operations.

Key Components

1. HAR Processor

Purpose: Parse and analyze HAR (HTTP Archive) files

Inputs:
- HAR file (JSON format)
- Endpoint filter rules
- Authentication patterns

Outputs:
- Parsed request/response pairs
- Endpoint inventory
- Static resource list
- Auth token candidates

Configuration Example:

yaml
 har-processor-config.yaml
processor:
  filters:
    static_resources:
      - ".css"
      - ".js"
      - ".png"
      - ".jpg"
      - ".woff2"

    hosts:
      include:
        - api.example.com
        - app.example.com
      exclude:
        - analytics.example.com

```
    methods:
      include:
        - GET
        - POST
        - PUT
        - DELETE

  auth_extraction:
    patterns:
      - "Bearer .+"
      - "Basic .+"
      - "Bearer .+"

    headers:
      - Authorization
      - X-API-Key
      - Cookie
```

---

2. Script Generator

Purpose: Convert processed HAR data into executable test scripts

Inputs:
- Processed HAR data
- Script template selection
- Configuration options

Outputs:
- JMeter (.jmx) XML
- Locust (.py) Python
- k6 (.js) JavaScript

Template Variables:

python
 Locust template example
TEMPLATE_VARIABLES = {
    "base_url": "https://api.example.com",
    "auth_token": "{{auth_token}}",
    "user_credentials": [
        {"username": "user1", "password": "pass1"},
        {"username": "user2", "password": "pass2"},
    ],
    "think_time": {
        "min": 1,
        "max": 3
    },
    "headers": {
        "Content-Type": "application/json",
        "Accept": "application/json"
    }
}

---

3. Correlation Engine

Purpose: Automatically identify and handle dynamic values

Inputs:
- Script to correlate
- Sample response data
- Correlation mode (Passive/Suggest/Agent)

Outputs:

- Updated script with extractors
- Correlation report
- Validation results

Correlation Types Supported:

Type	Description	Example
JSON Path	Navigate JSON structures	$.data.user.id
Regex	Pattern matching	token":"([^"]+)"
CSS Selector	HTML element extraction	auth-token
XPath	XML/HTML navigation	//input[@name='token']/@value

---

4. AI Agents

George - JMeter Genius

Function: Answer questions and debug JMeter issues

API Usage:

```
python
 Using George agent
response = loadmagic.agents.george.ask(
    question="How do I handle JWT tokens in JMeter?",
    context={
        "script_type": "jmeter",
        "auth_method": "Bearer",
        "api_version": "v2"
    }
)
```

```
print(response.answer)
 "To handle JWT tokens in JMeter, you can use the Regular Expression
  Extractor to capture the token from the login response..."
```

Rupert - Regex Researcher

Function: Generate regex patterns for data extraction

API Usage:

```
python
 Using Rupert for regex generation
response = loadmagic.agents.rupert.generate_regex(
    sample_data='{"user_id": 12345, "token": "abc123def456"}',
    extract_field="token",
    context="json"
)

print(response.regex)
 "token":\s"([^"]+)"
```

Suzy - Script Scholar

Function: Generate and modify scripts

API Usage:

```
python
 Using Suzy for script generation
response = loadmagic.agents.suzy.generate_script(
    description="Create a script that validates response contains valid user ID",
    language="groovy",
    framework="jmeter"
)
```

print(response.script)
 // Validate user ID response
// Response assertion for valid ID pattern

Carrie - The Correlator

Function: Orchestrate correlation pipeline

API Usage:

python
 Using Carrie for correlation
result = loadmagic.agents.carrie.correlate(
    script=script_content,
    responses=sample_responses,
    mode="agent"   passive, suggest, or agent
)

print(result.correlations_found)
print(result.script_updated)

Quinn - QA Assessor

Function: Analyze and improve test quality

API Usage:

python
 Using Quinn for QA assessment
assessment = loadmagic.agents.quinn.assess(
    script=script_content,
    check_types=["assertions", "correlations", "configuration"],
    auto_fix=True
)

print(assessment.issues_found)
print(assessment.fixes_applied)
print(assessment.score) 0-100

---

5. Results Visualizer

Purpose: Display and analyze test execution results

Features:
- Real-time metrics
- Transaction grouping
- Error analysis
- Performance graphs

Data Format:

json
{
  "execution_id": "exec_123456",
  "timestamp": "2026-02-23T10:30:00Z",
  "duration": 600,
  "summary": {
    "total_requests": 50000,
    "total_errors": 45,
    "error_rate": 0.09,
    "avg_response_time": 245,
    "p50_response_time": 180,
    "p95_response_time": 520,
    "p99_response_time": 1200,
    "throughput": 83.3
  },

```
  "transactions": [
    {
      "name": "User Login",
      "requests": 10000,
      "avg_time": 350,
      "errors": 12
    }
  ]
}
```

Data Flow Diagrams

1. HAR to Script Flow

Browser LoadMagic Script
Recording AI Generation

Analysis

AI
Agents

Step-by-Step:

1. Browser Recording - User records actions in browser
2. HAR Export - Browser exports HTTP Archive
3. Upload - HAR file uploaded to LoadMagic.AI
4. Parse - HAR Processor analyzes content
5. Analyze - AI Agents identify patterns

6. Generate - Script Generator creates test script
7. Output - Final script delivered

2. Correlation Flow

Script Response Correlation
+ Data Analysis Engine

Dynamic Extractor Updated
Values Rules Script
Found Created Ready

3. Self-Healing Pipeline

Self-Healing Pipeline

Execute Validate Auto- Repeat
Script Results Repair Offender

Decision Logic

If validation passes: Script continues

If auto-repair possible: Auto-fix and retry

If recurring issue: Advanced repair mode

If unresolvable: Report to user

Updated Script

with Fixes

API Reference

Authentication

bash

Using API Key

```
curl -H "Authorization: Bearer YOUR_API_KEY" \
    https://api.loadmagic.ai/v1/projects
```

Using OAuth2

```
curl -H "Authorization: Bearer YOUR_OAUTH_TOKEN" \
    https://api.loadmagic.ai/v1/projects
```

Endpoints

Method	Endpoint	Description
GET	/api/v1/projects	List all projects
POST	/api/v1/projects	Create new project
GET	/api/v1/projects/{id}	Get project details
PUT	/api/v1/projects/{id}	Update project

DELETE	/api/v1/projects/{id}	Delete project
POST	/api/v1/scripts/generate	Generate script from HAR
POST	/api/v1/executions/start	Start test execution
GET	/api/v1/executions/{id}	Get execution status
GET	/api/v1/results/{id}	Get test results

Summary

Component	Function	Key Technology
HAR Processor	Parse browser recordings	JSON parsing
Script Generator	Create test scripts	Template engine
Correlation Engine	Handle dynamic data	AI + Regex
AI Agents	Automate tasks	LLM integration
Results Visualizer	Display metrics	WebSocket + Charts

Next: [Terminology Glossary](./terminology-glossary.md)

2.2 Terminology Glossary

Terminology Glossary

Overview

This glossary defines key terms used throughout LoadMagic.AI documentation to ensure consistent understanding across all users.

A

API Key

A unique identifier used to authenticate requests to the LoadMagic.AI API. Can be generated in the dashboard and revoked at any time.

Example: lm_api_key_xxxxxxxxxxxx

Assertion

A verification mechanism in load tests that checks whether a response meets expected criteria. Common assertions include:
- Response code assertion
- Response content assertion
- Response time assertion
- JSON path assertion

B

Baseline Test

An initial test run that establishes performance benchmarks for comparison with subsequent tests. Used to detect performance regressions.

BeanShell

A Java-based scripting language that can be used in JMeter for custom scripting. Now largely replaced by JSR223 with Groovy.

C

Cache

A temporary storage layer that stores frequently accessed data to improve response times. Common cache types:
- Browser cache
- CDN cache
- Application cache
- Database query cache

CI/CD Pipeline

Continuous Integration/Continuous Deployment - automated processes that build, test, and deploy code changes. LoadMagic.AI integrates with major CI/CD platforms.

Correlation

The process of extracting dynamic values from responses and passing them to subsequent requests. Essential for maintaining session state across test iterations.

Example: Extracting a session token from a login response and using it in subsequent API calls.

CSV Data Set Config

A JMeter component that reads data from CSV files to parameterize tests with multiple user credentials or test data.

D

Dynamic Value

A value in an application response that changes with each request or user session. Common examples:
- Session IDs
- Authentication tokens
- Timestamps
- CSRF tokens
- Random identifiers

E

Endpoint

A specific URL path in an API that handles a particular type of request. Also referred to as a route or URI.

Example: GET /api/users/123

Execution

A single run of a load test script. Contains configuration, results, and metadata about the test run.

Extractors

JMeter components that parse response data to extract values for use in subsequent requests. Types include:
- Regular Expression Extractor

- JSON Path Extractor
- CSS Selector Extractor
- XPath Extractor

F

Flow

A sequence of related HTTP requests that represent a user journey or business transaction in a load test.

Example: Login Browse Products Add to Cart Checkout

G

Groovy

A dynamic language for the Java Virtual Machine, widely used in JMeter JSR223 scripting.

H

HAR (HTTP Archive)

A JSON-format log of web browser interactions, containing all HTTP requests and responses. The standard format for recording user sessions for load testing.

Structure:
json
{
  "log": {

```
    "version": "1.2",
    "creator": { "name": "Chrome", "version": "120.0" },
    "entries": [
      {
        "startedDateTime": "2026-02-23T10:00:00Z",
        "request": {
          "method": "GET",
          "url": "https://api.example.com/users",
          "headers": [...]
        },
        "response": {
          "status": 200,
          "content": { "size": 1234, "mimeType": "application/json" }
        }
      }
    ]
  }
}
```

I

Iteration

A single complete execution of all tasks in a load test script by one virtual user.

J

JMeter

Apache JMeter - an open-source load testing tool originally developed for web application testing,

now supporting multiple protocols.

JSR223

A Java specification for scripting that allows custom code execution in JMeter. Supports multiple languages including Groovy, JavaScript, and Python.

K

k6

An open-source load testing tool developed by Grafana Labs, using JavaScript for scripting.

L

Load Profile

The configuration defining how load is applied during a test, including:
- Number of virtual users
- Ramp-up period
- Test duration
- Think time between requests

Locust

An open-source load testing tool written in Python, using a code-first approach to defining test scenarios.

M

Metrics

Quantitative measurements collected during test execution. Key metrics include:
- Response time
- Throughput
- Error rate
- Concurrent users

Middleware

Software that acts as a bridge between an operating system or database and applications. In load testing, often refers to:
- API gateways
- Authentication services
- Caching layers

P

Parameterization

The process of replacing hardcoded values in test scripts with variables that can take different values for each test iteration.

Pre-Processor

A JMeter element that runs before a sampler, often used for:
- Setting up variables
- Generating dynamic data

- Modifying requests

Post-Processor

A JMeter element that runs after a sampler, typically used for:
- Extracting data from responses
- Processing correlation values
- Data validation

R

Ramp-up Period

The time period over which virtual users are gradually added to reach the target load level.

Example: 10,000 users over 10 minutes = 1,000 users/minute ramp-up

---

Regex (Regular Expression)

A pattern matching language used to extract data from text responses. Essential for correlation in load testing.

Example: session_id=([a-zA-Z0-9]+) extracts alphanumeric session IDs

---

Request

An HTTP message sent from client to server, containing:
- Method (GET, POST, PUT, DELETE, etc.)
- URL

- Headers
- Body (optional)

Response

An HTTP message returned from server to client, containing:
- Status code
- Headers
- Body (HTML, JSON, XML, etc.)

S

Sampler

A JMeter element that generates an HTTP request. Types include:
- HTTP Request
- JDBC Request
- FTP Request
- SOAP/XML-RPC Request

Script

A load test definition file containing:
- Request definitions
- Correlation logic
- Assertions
- Configuration

Self-Healing

LoadMagic.AI's ability to automatically detect and fix correlation issues without human intervention.

Session

A period of activity by a single virtual user, typically including multiple requests.

Static Resource

Files that don't change between requests and don't require correlation:
- CSS files
- JavaScript files
- Images
- Fonts

Stress Testing

A type of load test that pushes the system beyond normal operational capacity to find breaking points.

T

Test Plan

A complete definition of a load test in JMeter, containing:
- Thread groups
- Controllers

- Samplers
- Listeners
- Configuration elements

Think Time

Delays between user actions to simulate realistic human behavior. Can be fixed or randomized.

Thread Group

The foundation element in JMeter that defines:
- Number of virtual users
- Ramp-up time
- Loop count
- Scheduler configuration

Throughput

The number of requests processed per unit of time, typically measured in:
- Requests per second (RPS)
- Requests per minute (RPM)
- Transactions per second (TPS)

Transaction

A logical grouping of one or more requests representing a complete user action.

V

Virtual User (VU)

A simulated user executing test scenarios. Each virtual user runs independently with its own session data.

W

Workload Model

A representation of expected real-world usage patterns, including:
- User distribution
- Transaction mix
- Peak periods
- Usage patterns

Summary

This glossary covers the essential terms for working with LoadMagic.AI. For more detailed information on any term, refer to the relevant documentation section.

Next: [Diagrams](./diagrams.md)

2.3 Diagrams

Diagrams

Overview

This document describes the key diagrams used throughout LoadMagic.AI documentation. Each diagram includes a text-based representation and guidance for creating visual versions.

---

1. System Architecture Diagram

Description

High-level overview of the LoadMagic.AI platform architecture showing the major components and their interactions.

Text Representation

LoadMagic.AI Platform

Client Layer

Web UI (Studio) JMeter Plugin REST API

API Gateway Layer

Auth & Security Rate Limiting Request Routing

Core Services Layer

HAR Script Correlation
Processing Generation Engine

AI Results User
Agents Visualizer Management

Data & Storage Layer

Document AI Results
Store Model Store
(MongoDB) Storage (S3/GCS)

Visual Version Guidelines

Recommended Tools: Draw.io, Lucidchart, Mermaid.js

Layout:
- Layered top-to-bottom approach
- Three main tiers: Client, Processing, Data

- Clear boundaries between layers
- Color coding by component type

---

2. AI Agents Interaction Diagram

Description

Shows how the five AI agents work together to automate performance testing.

Text Representation

AI Agents Collaboration

User

Request

Agent Orchestration Layer

Request Router

George	Rupert	Suzy
(Expert)	(Regex)	(Script)
Debug	Regex	Generate
Answer	JSON	Convert
Explain	Extract	Fix

Carrie
(Correlator)

Quinn
(QA)

Visual Version Guidelines

Relationship Types:
- Solid lines: Direct interaction
- Dotted lines: Delegation
- Bidirectional arrows: Feedback loops

Color Coding:
- George: Blue (knowledge)

- Rupert: Green (extraction)
- Suzy: Purple (creation)
- Carrie: Orange (orchestration)
- Quinn: Teal (quality)

---

3. Data Processing Pipeline

Description

Shows how data flows through the LoadMagic.AI system from HAR file to results.

Text Representation

Browser Recording → HAR Processor → Script Generator → Results Visualizer

AI Agents
Correlation Engine
Performance Metrics

Mermaid Code

```
mermaid
graph LR
    A[Browser Recording] --> B[HAR Processor]
    B --> C[Script Generator]
    C --> D[Results Visualizer]

    B --> E[AI Agents]
    C --> F[Correlation Engine]
```

```
    D --> G[Performance Metrics]

    style A fill:e1f5fe
    style B fill:e1f5fe
    style C fill:e1f5fe
    style D fill:e1f5fe
    style E fill:fff3e0
    style F fill:fff3e0
    style G fill:e8f5e9
```

---

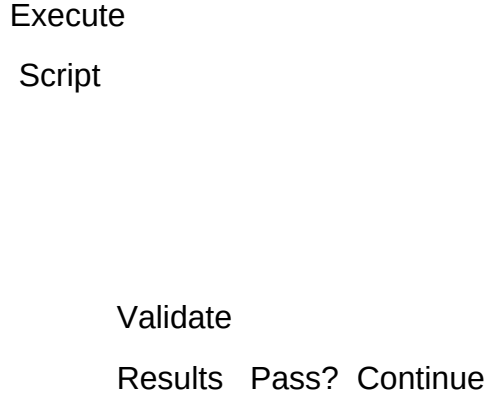
4. Correlation Self-Healing Flow

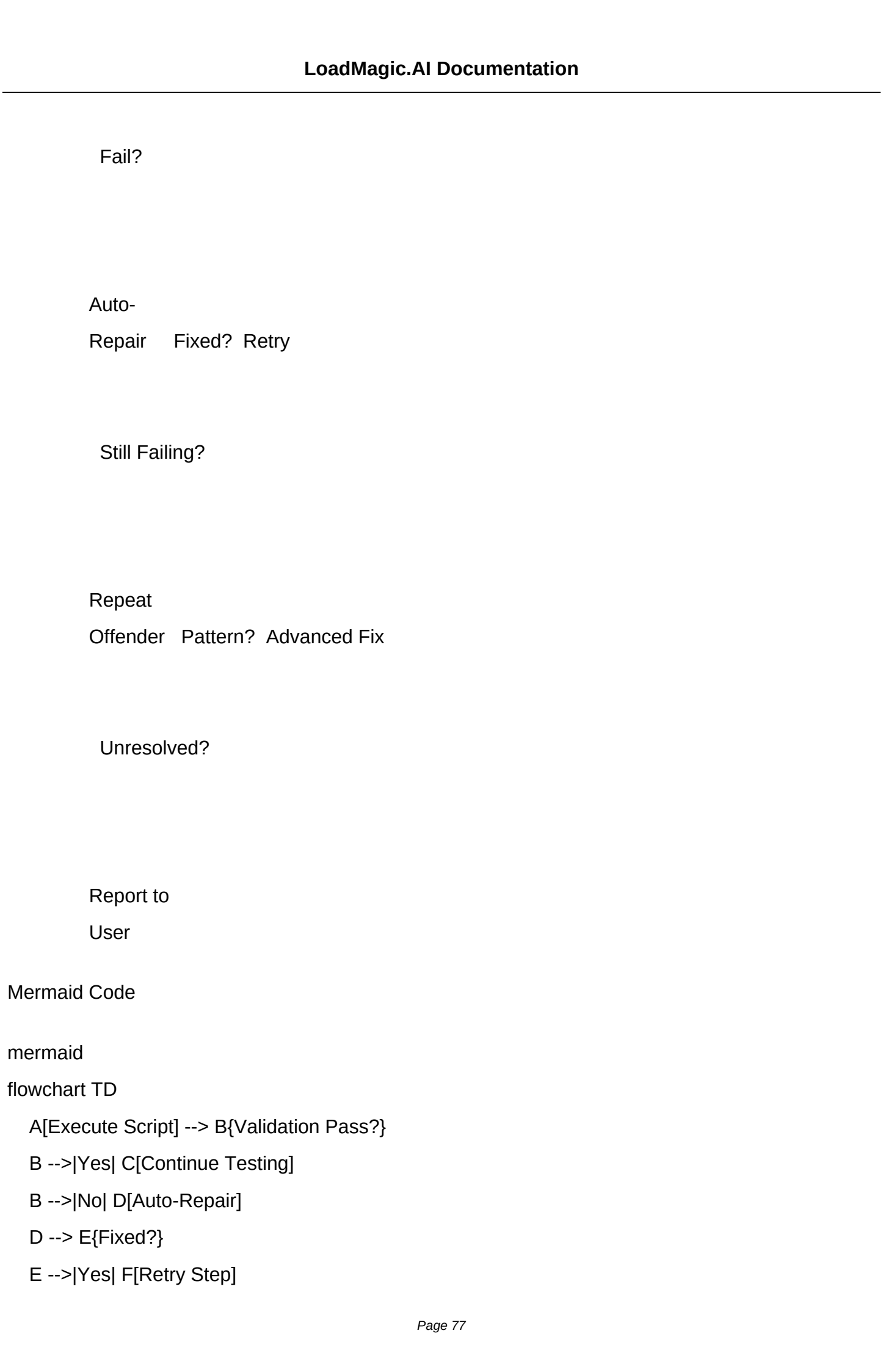
Description

Illustrates the 5-phase self-healing pipeline for automated correlation.

Text Representation

Self-Healing Pipeline





Mermaid Code

```
mermaid
flowchart TD
    A[Execute Script] --> B{Validation Pass?}
    B -->|Yes| C[Continue Testing]
    B -->|No| D[Auto-Repair]
    D --> E{Fixed?}
    E -->|Yes| F[Retry Step]
```

```
E -->|No| G{Repeat Offender?}
G -->|Yes| H[Advanced Repair]
G -->|No| I[Missed Candidates]
H --> F
I --> F
F --> B

style A fill:ffcdd2
style B fill:fff9c4
style C fill:c8e6c9
style D fill:ffccbc
style E fill:fff9c4
style F fill:bbdefb
style G fill:fff9c4
style H fill:ffccbc
style I fill:ffccbc
```

---

5. Enterprise Integration Diagram

Description

Shows how LoadMagic.AI integrates with enterprise systems and workflows.

Text Representation

Enterprise Integration

LoadMagic.AI Core

Script Store | Execution Engine | Results Store | AI Agents

CI/CD | Monitoring | SSO Provider

Jenkins | Prometheus |
GitLab | Grafana | Okta
GitHub | Datadog | Azure AD
Azure | New Relic | Google

Source Control | Metrics & Logs | Identity & Access

GitHub | CloudWatch | Users
Bitbucket | Stackdriver | Groups
GitLab | ELK Stack | Roles

---

6. Load Profile Diagram

Description

Visual representation of a typical load test profile showing virtual user ramp-up and sustain phases.

Text Representation

Users

10K

Time

0 5 10 15 20 25 30 35 40

Ramp-up Sustain Ramp-down End

Peak Load

Mermaid Code

mermaid

xychart-beta

title "Load Test Profile"

x-axis "Time (minutes)" --> [0, 5, 10, 15, 20, 25, 30, 35, 40]

y-axis "Virtual Users" --> [0, 2000, 4000, 6000, 8000, 10000]

line [0, 1000, 3000, 6000, 9000, 10000, 10000, 10000, 5000, 0]

---

7. Test Execution Flow

Description

Shows the lifecycle of a load test execution from start to results analysis.

Text Representation

Test Execution Lifecycle

1. PREPARE

Script Config Validate Queue
Selection Settings Inputs Request

2. EXECUTE

Spawn Inject Execute Monitor
VUs Load Requests Metrics

3. COLLECT

Aggregate Store Analyze Generate
Results Results Metrics Reports

4. ANALYZE

Compare Identify Recommend Archive
Baseline Issues Fixes Results

Summary

These diagrams provide visual representations of key LoadMagic.AI concepts and workflows. For each diagram:

Diagram	Purpose	Tool Recommendation
System Architecture	Platform overview	Draw.io, Lucidchart
AI Agents	Agent collaboration	Mermaid, Draw.io
Data Pipeline	Processing flow	Mermaid
Self-Healing	Correlation automation	Mermaid, flowcharts
Enterprise Integration	External systems	Draw.io
Load Profile	Test configuration	Excel, Mermaid
Test Execution	Lifecycle stages	Flowchart tools

Next: [Quickstart Guide](../3-Quickstart/setup-options.md)

3. Quickstart

Setup Options

Overview

LoadMagic.AI offers multiple deployment options to meet different organizational needs. This guide covers all setup methods from quick cloud trials to enterprise on-premises deployments.

Option 1: Cloud (SaaS)

Quick Start

The fastest way to get started with LoadMagic.AI.

Steps:

1. Sign Up
   - Visit https://loadmagic.ai
   - Click "Try FREE"
   - Enter email and password
   - Verify email address

2. Access Dashboard
   - Navigate to https://studio.loadmagic.ai
   - Log in with credentials
   - View the Locust AI Studio interface

3. Start Testing
   - Upload your first HAR file
   - Generate your first script
   - Run your first test

Supported Regions

Region	Endpoint	Latency (avg)
US East	us-east-1.loadmagic.ai	20ms
US West	us-west-2.loadmagic.ai	45ms

EU (Frankfurt)	eu-central-1.loadmagic.ai	80ms
UK (London)	eu-west-2.loadmagic.ai	70ms
APAC (Singapore)	ap-southeast-1.loadmagic.ai	150ms

Environment Variables

bash
 Cloud configuration
LOADMAGIC_API_URL=https://api.loadmagic.ai
LOADMAGIC_REGION=us-east-1
LOADMAGIC_AUTH_TYPE=api_key

Option 2: JMeter Plugin (Desktop)

Prerequisites

Requirement	Minimum	Recommended
Java	OpenJDK 11+	OpenJDK 17+
JMeter	5.5+	5.7+
RAM	4GB	8GB+
Disk	2GB	10GB
OS	Windows/macOS/Linux	Windows 10+/macOS 12+

Installation Steps

Windows

1. Download JMeter
   powershell
    Using Chocolatey
   choco install jmeter

Or manually from https://jmeter.apache.org/

2. Download LoadMagic Plugin
   - Visit https://loadmagic.ai/more
   - Download LoadMagic-Plugin-.jar

3. Install Plugin
   powershell
    Copy to JMeter lib/ext directory
   Copy-Item LoadMagic-Plugin-.jar $JMETER_HOME\lib\ext\

4. Launch JMeter
   powershell
    Start JMeter with LoadMagic
   $env:LOADMAGIC_API_KEY="your_api_key"
   jmeter.bat

macOS

bash
 Install JMeter
brew install jmeter

Download plugin
curl -L -o ~/jmeter/lib/LoadMagic-Plugin-.jar \
  "https://loadmagic.ai/download/LoadMagic-Plugin-.jar"

Set environment
export LOADMAGIC_API_KEY="your_api_key"

Launch
open /usr/local/bin/jmeter

Linux

bash
 Install Java and JMeter
sudo apt-get update
sudo apt-get install openjdk-11-jre
wget https://archive.apache.org/dist/jmeter/binaries/apache-jmeter-5.7.tgz
tar -xzf apache-jmeter-5.7.tgz

Download plugin
wget -O ~/jmeter/lib/LoadMagic-Plugin-.jar \
  "https://loadmagic.ai/download/LoadMagic-Plugin-.jar"

Set environment
export LOADMAGIC_API_KEY="your_api_key"
export JMETER_HOME=~/apache-jmeter-5.7

Launch
$JMETER_HOME/bin/jmeter

Configuration

properties
 jmeter.properties additions
loadmagic.api.key=your_api_key_here
loadmagic.auto.save=true
loadmagic.debug.mode=false
loadmagic.agent.mode=passive

Option 3: REST API (Programmatic Access)

For Developers

Access LoadMagic.AI programmatically for automation and CI/CD integration.

Python SDK Installation

bash
 Install via pip
pip install loadmagic-sdk

Or install from source
git clone https://github.com/loadmagic/loadmagic-python.git
cd loadmagic-python
pip install -e .

Node.js SDK Installation

bash
 Install via npm
npm install @loadmagic/sdk

Or install from source
git clone https://github.com/loadmagic/loadmagic-node.git
cd loadmagic-node
npm install

JavaScript/TypeScript Setup

bash
 Install via npm
npm install @loadmagic/client

API Key Generation

1. Log in to dashboard
2. Navigate to Settings API Keys
3. Click Generate New Key

4. Copy and securely store the key

Quick Test

python
 Python example
from loadmagic import LoadMagic

Initialize client
lm = LoadMagic(api_key="your_api_key")

Test connection
status = lm.health.check()
print(f"Status: {status}")

javascript
// JavaScript example
const { LoadMagic } = require('@loadmagic/client');

const lm = new LoadMagic({
  apiKey: 'your_api_key'
});

// Test connection
const status = await lm.health.check();
console.log('Status:', status);

Option 4: Enterprise (Private Cloud / On-Premises)

For Enterprise Customers

Contact sales@loadmagic.ai for enterprise options.

Requirements

Kubernetes (Recommended)

Component	Requirement
Kubernetes	1.24+
Nodes	3+ (for HA)
CPU	16+ cores total
RAM	64GB+ total
Storage	500GB+ SSD
Network	1Gbps+

Docker Standalone

bash
 Minimum requirements
CPU: 8 cores
RAM: 32GB
Disk: 200GB SSD

Installation

Option A: Helm Chart

bash
 Add LoadMagic helm repository
helm repo add loadmagic https://charts.loadmagic.ai
helm repo update

Install with custom values
helm install loadmagic loadmagic/loadmagic \
  --namespace loadmagic \
  --create-namespace \
  --values values-prod.yaml

Option B: Docker Compose

bash
 Clone installation repo
git clone https://github.com/loadmagic/loadmagic-deploy.git
cd loadmagic-deploy

Configure environment
cp .env.example .env
 Edit .env with your settings

Start services
docker-compose up -d

Option C: Kubernetes Manifests

bash
 Apply manifests
kubectl apply -f k8s/namespace.yaml
kubectl apply -f k8s/configmap.yaml
kubectl apply -f k8s/secrets.yaml
kubectl apply -f k8s/deployments/
kubectl apply -k k8s/services/

Configuration Example

yaml
 values-prod.yaml
global:
  domain: "loadmagic.yourcompany.com"
  storageClass: "fast-ssd"

ingress:
  enabled: true

```
  className: "nginx"
  annotations:
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
  hosts:
    - host: "loadmagic.yourcompany.com"
      paths:
        - path: /
          pathType: Prefix

database:
  type: "postgresql"
  host: "your-db-host.internal"
  port: 5432
  name: "loadmagic"
  poolSize: 20

redis:
  host: "your-redis.internal"
  port: 6379

s3:
  endpoint: "https://s3.yourregion.amazonaws.com"
  bucket: "loadmagic-data"
  region: "your-region"

loadmagic:
  replicas: 3
  resources:
    limits:
      cpu: "2000m"
      memory: "4Gi"
    requests:
      cpu: "1000m"
```

```
    memory: "2Gi"
```

Environment Comparison

Feature	Cloud	Plugin	API	Enterprise
Setup time	5 min	30 min	15 min	1-2 days
Cost	Free tier available	Free	Usage-based	Custom
AI Agents				
Auto-correlation				
SSO/SAML				
On-premises				
SLA	99.9%	N/A	99.9%	99.99%
Support	Email	Email	Email	24/7 + Phone

Choosing Your Setup

Use Case	Recommended Option
Individual testing	Cloud (Free tier)
Team collaboration	Cloud (Pro)
Local development	JMeter Plugin
CI/CD automation	REST API
Enterprise security	Private Cloud
Regulatory compliance	On-premises

Next Steps

- [Workspace Basics](./workspace-basics.md) - Learn the interface
- [Hello-World Pipeline](./hello-world-pipeline.md) - Run your first test

---

Next: [Workspace & CLI Basics](./workspace-basics.md)

3.1 Workspace & CLI Basics

Workspace & CLI Basics

Overview

This guide covers the LoadMagic.AI workspace interface and command-line tools. You'll learn to navigate the platform and execute common operations.

Web Interface (Locust AI Studio)

Getting Started

After logging in, you'll see the main dashboard:

LoadMagic.AI [User] [Logout]

Projects New Test Analytics
icon icon icon
View all Create Trends

Recent Projects

Project Name	Last Modified	Status
E-commerce API	2 hours ago	Active
User Authentication	Yesterday	Active
Payment Gateway	3 days ago	Draft

Main Features

1. Project Management

yaml
Project Operations:
- Create new project
- Import existing scripts
- Clone project
- Export scripts
- Delete project
- Share with team

2. Script Editor

File Project DL Editor: api-test.py [Run] [Save]

Files
api-test
api-t
user
helper
data

```
1  from locust import HttpUser, task
2
3  class APIUser(HttpUser):
4      wait_time = (1, 3)
5
6      @task
```

```
creds     7     def get_users(self):
config    8         self.client.get("/api/users")
```

Carrie is ready. [Passive] [Send message...]

3. Results Panel

Results [Download CSV] [Export Report]

Summary Details

Total Requests: 50,000 Response Time Distribution
Success Rate: 99.2%
Avg Response: 245ms
Peak Concurrent: 1,000 p50:180ms p95:520ms p99:1s
Duration: 10 min

Errors:
- 401 Unauthorized: 45 (0.09%)
- 500 Server Error: 12 (0.02%)

CLI Tools

Installation

bash
Python CLI
pip install loadmagic-cli

Verify installation

lm --version
 Output: loadmagic-cli 1.2.7

Configuration

bash
 Initialize configuration
lm init

This creates ~/.loadmagic/config.yaml

yaml
 ~/.loadmagic/config.yaml
api:
  url: https://api.loadmagic.ai
  key: your_api_key_here
  region: us-east-1

defaults:
  project: default
  output: ./results
  format: json

Common Commands

1. Project Management

bash
 Create new project
lm project create --name "api-load-test" --description "API performance testing"

List projects
lm project list

Get project details

lm project info api-load-test

Delete project

lm project delete api-load-test

2. Script Operations

bash

Generate script from HAR

lm script generate --har ./recordings/shop-flow.har \

--output ./scripts/shop-flow.py \

--framework locust

Validate script syntax

lm script validate ./scripts/shop-flow.py

Import existing script

lm script import ./tests/jmeter/test-plan.jmx

3. Execution

bash

Run test

lm run ./scripts/shop-flow.py \

--users 1000 \

--duration 10m \

--spawn-rate 100

Run with custom config

lm run ./scripts/shop-flow.py \

--config ./configs/peak-load.yaml

4. Results

bash

List results

```
lm results list --project api-load-test
```

Get result details

```
lm results get exec_123456
```

Download results

```
lm results download exec_123456 --output ./results/
```

Compare results

```
lm results compare exec_123456 exec_789012
```

Configuration Files

Run Configuration

yaml

peak-load.yaml

```
execution:
  users: 5000
  spawn_rate: 500
  duration: 30m
  run_on: all_nodes

targets:
  - host: api.example.com
    protocol: https
    port: 443

thresholds:
  avg_response_time: 500ms
  p95_response_time: 1000ms
  error_rate: 1%
```

```
reporting:
  results_file: results.jtl
  dashboard: true
  notifications:
    - email: team@example.com
    - slack: "performance-alerts"
```

Python SDK Usage

Basic Operations

```
python
from loadmagic import LoadMagic

Initialize
lm = LoadMagic(api_key="your_api_key")

Create project
project = lm.projects.create(
    name="API Load Test",
    description="Performance testing for REST API"
)

Generate script
script = lm.scripts.generate(
    har_file="recordings/login-flow.har",
    framework="locust",
    options={
        "correlation": "auto",
        "extract_auth": True
    }
)
```

```
Execute test
execution = lm.executions.start(
    script_id=script.id,
    config={
        "users": 1000,
        "duration": "10m"
    }
)

Monitor execution
while execution.status == "running":
    execution = lm.executions.get(execution.id)
    print(f"Progress: {execution.progress}%")

 Get results
results = lm.results.get(execution.id)
print(f"Avg Response: {results.avg_response_time}ms")
```

JavaScript/TypeScript SDK

Basic Operations

```
typescript
import { LoadMagic, Locust } from '@loadmagic/client';

// Initialize
const lm = new LoadMagic({
  apiKey: process.env.LOADMAGIC_API_KEY,
  region: 'us-east-1'
});

// Create project
```

```
const project = await lm.projects.create({
  name: 'API Load Test',
  description: 'Performance testing'
});

// Generate script
const script = await lm.scripts.generate({
  harFile: './recordings/login-flow.har',
  framework: 'locust',
  options: {
    correlation: 'auto',
    extractAuth: true
  }
});

// Execute test
const execution = await lm.executions.start({
  scriptId: script.id,
  config: {
    users: 1000,
    duration: '10m'
  }
});

// Stream results
const stream = await lm.executions.stream(execution.id);
stream.on('data', (data) => {
  console.log('Progress:', data.progress);
});
```

Environment Variables

Variable	Description	Required
LOADMAGIC_API_KEY	Your API key	Yes
LOADMAGIC_API_URL	API endpoint	No
LOADMAGIC_REGION	AWS region	No
LOADMAGIC_DEBUG	Enable debug mode	No

Summary

Interface	Best For	Learning Curve
Web UI	Visual editing, exploration	Low
CLI	Automation, CI/CD	Medium
Python SDK	Custom integrations	Medium
JS/SDK	Node.js applications	Medium

Next: [Hello-World Pipeline](./hello-world-pipeline.md)

3.2 Hello-World Pipeline

Hello-World Pipeline

Overview

This guide walks you through creating your first load test pipeline with LoadMagic.AI. By the end, you'll have a complete working test that you can run and analyze.

Estimated Time: 15-20 minutes
Prerequisites: LoadMagic.AI account (free tier works)

Step 1: Record a HAR File

Using Chrome

1. Open Chrome DevTools
   - Press F12 or Cmd+Option+I (macOS)
   - Click the Network tab

2. Start Recording
   - Ensure the record button (red circle) is active
   - Check Preserve log option

3. Perform Test Actions
   - Navigate to your target website
   - Log in (if required)
   - Perform key user flows

4. Export HAR
   - Right-click in the Network panel
   - Select Save all as HAR with content
   - Save as login-flow.har

Using Firefox

1. Open Developer Tools
   - Press F12 or Cmd+Option+K (macOS)
   - Click the Network tab

2. Record Session
   - Click the gear icon
   - Enable Persist Logs

3. Perform Actions and Export

- Do your test actions
- Click the gear Save All as HAR

Sample HAR Structure

Your HAR file should look like this:

```
json
{
  "log": {
    "version": "1.2",
    "creator": {
      "name": "Chrome",
      "version": "120.0.0"
    },
    "entries": [
      {
        "startedDateTime": "2026-02-23T10:00:00.000Z",
        "time": 245.123,
        "request": {
          "method": "POST",
          "url": "https://api.example.com/auth/login",
          "httpVersion": "HTTP/1.1",
          "headers": [
            {"name": "Content-Type", "value": "application/json"},
            {"name": "User-Agent", "value": "..."}
          ],
          "queryString": [],
          "postData": {
            "mimeType": "application/json",
            "text": "{\"username\":\"test\",\"password\":\"test123\"}"
          }
```

```
      },
      "response": {
        "status": 200,
        "statusText": "OK",
        "headers": [
          {"name": "Content-Type", "value": "application/json"},
          {"name": "Authorization", "value": "Bearer eyJhbGciOiJIUzI1..."}
        ],
        "content": {
          "mimeType": "application/json",
          "text": "{\"userId\":12345,\"token\":\"eyJhbGciOiJIUzI1NiJ9...\"}"
        }
      }
    }
  ]
 }
}
```

Step 2: Upload to LoadMagic.AI

Using Web Interface

1. Navigate to Locust AI Studio
   - Go to https://studio.loadmagic.ai
   - Log in

2. Create or Open Project
   - Click New Project
   - Name it "Hello World Test"
   - Click Create

3. Upload HAR
   - Click the Drop/Browse HAR button
   - Select your login-flow.har file
   - Wait for upload and processing

4. Configure Options

   Endpoint filters: All
   Process HAR: Click
   Attach transaction markers: Optional

Using CLI

bash
 Upload and process HAR
lm har upload ./login-flow.har \
  --project "Hello World Test" \
  --process

 List processed endpoints
lm har endpoints login-flow.har

Step 3: Generate Script

Automatic Generation

After uploading, LoadMagic.AI automatically:

1. Parses the HAR file
2. Identifies endpoints and methods
3. Extracts authentication patterns
4. Creates a Locust script

Generated Script Example

```
python
 Generated by LoadMagic.AI
from locust import HttpUser, task, between, events
import json

class HelloWorldUser(HttpUser):
    wait_time = between(1, 3)

    def on_start(self):
        """Called when a simulated user starts."""
        response = self.client.post("/auth/login", json={
            "username": "test",
            "password": "test123"
        })

        if response.status_code == 200:
            data = response.json()
            self.token = data.get("token")
        else:
            self.token = None

    @task(3)
    def get_user_profile(self):
        """Get user profile - most common action."""
        if self.token:
            self.client.get(
                "/api/user/profile",
                headers={"Authorization": f"Bearer {self.token}"}
            )
```

```
    @task(2)
    def get_dashboard(self):
        """Load user dashboard."""
        if self.token:
            self.client.get(
                "/api/dashboard",
                headers={"Authorization": f"Bearer {self.token}"}
            )

    @task(1)
    def get_notifications(self):
        """Check notifications."""
        if self.token:
            self.client.get(
                "/api/notifications",
                headers={"Authorization": f"Bearer {self.token}"}
            )
```

Manual Configuration

You can customize the generated script:

```
python
 Custom configuration
class HelloWorldUser(HttpUser):
     Customize wait time
    wait_time = between(0.5, 2)   More aggressive

     Add weight to prioritize tasks
    @task(10)   Higher weight = more frequent
    def get_dashboard(self):
        self.client.get("/api/dashboard")
```

```
    Add heavy endpoint
    @task(1)
    def search_products(self):
        self.client.get("/api/products?page=1&limit=50")
```

Step 4: Run Test

Using Web Interface

1. Configure Load Parameters

   Users: 100
   Spawn Rate: 10 users/second
   Duration: 5 minutes

2. Start Execution
   - Click the Run button
   - Watch the real-time results

3. Monitor Progress

   Progress: 45%
   Active Users: 45
   Requests/sec: 125
   Avg Response: 180ms
   Errors: 0

Using CLI

bash
 Run test with custom parameters

```
lm run ./hello-world.py \
  --users 100 \
  --spawn-rate 10 \
  --duration 5m \
  --headless

 Or with config file
lm run ./hello-world.py --config test-config.yaml

yaml
 test-config.yaml
execution:
  users: 100
  spawn_rate: 10
  duration: 5m
  run_on: single

targets:
  - host: api.example.com
    protocol: https
    port: 443

Using Python SDK

python
from loadmagic import LoadMagic

lm = LoadMagic(api_key="your_api_key")

Start execution
execution = lm.executions.start(
    script="./hello-world.py",
    config={
```

```
        "users": 100,
        "spawn_rate": 10,
        "duration": "5m",
        "host": "api.example.com"
    }
)

Wait for completion
execution.wait()

Get results
results = execution.results
print(f"Total Requests: {results.total_requests}")
print(f"Success Rate: {results.success_rate}%")
print(f"Avg Response: {results.avg_response_time}ms")
```

Step 5: Analyze Results

Results Dashboard

Test Results

Summary

Total Requests: 15,234
Success Rate: 99.7%
Avg Response: 187ms
Peak Users: 100
Duration: 5 minutes

Response Time Distribution

p50: 145ms
p90: 280ms
p95: 420ms
p99: 890ms

Throughput

Peak: 52 req/s
Average: 50 req/s

Errors

401 Unauthorized: 45 (0.3%)
- All from /api/user/profile
- Likely token expiration

Download Results

bash
Download JTL file
lm results download exec_abc123 --format jtl --output ./

Download CSV
lm results download exec_abc123 --format csv --output ./

Download HTML report
lm results download exec_abc123 --format html --output ./

Key Metrics Explained

| Metric | Definition | Target |

|--------|------------|--------|
Total Requests	Number of HTTP requests made	Depends on load
Success Rate	% of requests with 2xx/3xx response	> 99%
Avg Response	Mean response time	< 500ms
p95 Response	95th percentile	< 1000ms
p99 Response	99th percentile	< 2000ms
Throughput	Requests per second	Business target

Complete Example

Full Pipeline Script (Python)

python
!/usr/bin/env python3
"""
Hello World Load Test Pipeline
Run with: python hello_world_pipeline.py
"""

from loadmagic import LoadMagic
import time

def main():
    Initialize
    lm = LoadMagic(api_key="your_api_key")

    Step 1: Upload HAR
    print(" Uploading HAR file...")
    har_result = lm.hars.upload("./login-flow.har")
    print(f"   Processed {len(har_result.endpoints)} endpoints")

```
 Step 2: Generate script
print(" Generating Locust script...")
script = lm.scripts.generate(
    har=har_result.id,
    framework="locust"
)
print(f"    Script created: {script.name}")

 Step 3: Run test
print(" Starting load test...")
print("   Configuration: 100 users, 5 minutes")

execution = lm.executions.start(
    script_id=script.id,
    config={
        "users": 100,
        "spawn_rate": 10,
        "duration": "5m",
        "host": "api.example.com"
    }
)

 Monitor progress
while execution.status == "running":
    progress = lm.executions.progress(execution.id)
    print(f"   Progress: {progress.percent}% | Users: {progress.active_users}")
    time.sleep(10)

 Step 4: Get results
print(" Retrieving results...")
results = lm.results.get(execution.id)
```

```
    print("\n" + "="*50)
    print("         TEST RESULTS SUMMARY")
    print("="*50)
    print(f"Total Requests:  {results.total_requests:,}")
    print(f"Success Rate:    {results.success_rate:.1f}%")
    print(f"Avg Response:    {results.avg_response_time}ms")
    print(f"p95 Response:    {results.p95_response_time}ms")
    print(f"Throughput:      {results.throughput:.1f} req/s")
    print("="*50)

     Export results
    lm.results.export(execution.id, format="html", output="./report.html")
    print("\n Report saved to: ./report.html")

if __name__ == "__main__":
    main()
```

Troubleshooting

Common Issues

Issue	Cause	Solution
No requests generated	Empty HAR file	Re-record with actual traffic
Auth not working	Token extraction failed	Check HAR has login response
Script errors	Invalid Python syntax	Review generated code
Connection timeout	Target not accessible	Verify host/port
High error rate	Target overloaded	Reduce user load

Debug Mode

bash
 Enable debug logging
export LOADMAGIC_DEBUG=true
lm run ./test.py --verbose

Next Steps

Now that you've completed your first pipeline:

- [Data Connectors](../4-Core-Usage/data-connectors.md) - Learn about HAR processing
- [Pipeline Orchestration](../4-Core-Usage/pipeline-orchestration.md) - Build complex flows
- [CI/CD Integration](../6-Enterprise-Implementation/cicd-pipelines.md) - Automate in your pipeline

---

Next: [Common Troubleshooting](./troubleshooting.md)

3.3 Common Troubleshooting

Common Troubleshooting

Overview

This guide covers common issues encountered when using LoadMagic.AI and their solutions. Use this reference to quickly diagnose and resolve problems.

Authentication Issues

Invalid API Key

Symptoms:

- 401 Unauthorized error
- "Invalid API key" message

Solution:

bash
 Verify your API key
lm config show

Regenerate key if compromised
lm auth regenerate-key

Update configuration
lm config set api.key your_new_key

Token Expiration

Symptoms:
- 401 Unauthorized after some time
- "Token expired" message

Solution:

python
 Refresh token automatically
lm = LoadMagic(
    api_key="your_key",
    auto_refresh=True
)

Or manually refresh
lm.auth.refresh_token()

HAR File Issues

Empty HAR File

Symptoms:
- "No entries found" error
- Zero endpoints processed

Causes:
- Recording didn't capture traffic
- Wrong export format used

Solution:

bash
 Verify HAR file has content
cat your-file.har | jq '.log.entries | length'

Re-record using Chrome:
 1. Open DevTools (F12)
 2. Network tab
 3. Perform actions
 4. Right-click Save all as HAR with content

HAR File Too Large

Symptoms:
- Upload fails or times out
- "File too large" error

Solution:

bash
 Split large HAR files
lm har split large.har --max-size 10MB

Or filter before upload

lm har filter large.har \

  --include-endpoints "/api/" \

  --exclude-endpoints ".js,.css,.png"

Unsupported Format

Symptoms:

- "Invalid HAR format" error

- Parsing failures

Solution:

bash

 Validate HAR structure

lm har validate your-file.har

Convert from other formats

 Chrome: DevTools Network Right-click Save as HAR

 Firefox: DevTools Network Settings Save all as HAR

 Charles Proxy: Export HAR

Script Generation Issues

Correlation Failures

Symptoms:

- Script runs but fails with auth errors

- 401/403 errors on subsequent requests

- Session not maintained

Solution:

```
python
 Enable auto-correlation
script = lm.scripts.generate(
    har_file="flow.har",
    correlation="auto"
)

Manually add correlation if needed
 In generated script, add:
class APIUser(HttpUser):
    def on_start(self):
         Extract token from login response
        response = self.client.post("/login", json={...})
        self.token = response.json()["token"]

    @task
    def authenticated_request(self):
        self.client.get(
            "/api/data",
            headers={"Authorization": f"Bearer {self.token}"}
        )

Dynamic Value Not Extracted

Symptoms:
- Wrong data in requests
- IDs not matching expected patterns

Solution:

yaml
 Configure extraction patterns
correlation:
```

```
  patterns:
    - name: user_id
      type: json_path
      expression: "$.user.id"
      from_response: "/api/login"

    - name: session_id
      type: regex
      pattern: "sessionId=([^;]+)"
      from_response: "/api/init"
```

Execution Issues

Target Unreachable

Symptoms:
- Connection timeout errors
- "Host unreachable" message

Troubleshooting Steps:

bash
 1. Verify target is accessible
ping api.example.com

2. Check port is open
nc -zv api.example.com 443

3. Test with curl
curl -I https://api.example.com/health

4. Check firewall rules

Ensure outbound traffic allowed to target

High Error Rate

Symptoms:
- Error rate > 5%
- Many 5xx responses

Diagnosis:

python
Check error distribution
results = lm.results.get(execution_id)
for error in results.errors:
    print(f"{error.status_code}: {error.count} occurrences")
    print(f" Endpoints: {error.endpoints}")
    print(f" Sample: {error.sample}")

Common Causes & Solutions:

Error	Cause	Solution
500 Internal Server	Application bug	Check server logs
502 Bad Gateway	Proxy issue	Check upstream services
503 Service Unavailable	Overload	Reduce load
504 Gateway Timeout	Slow response	Increase timeout

Out of Memory

Symptoms:
- JMeter crashes
- "OutOfMemoryError" in logs

Solution:

bash
 Increase heap size
export JVM_ARGS="-Xmx4g -Xms2g"
jmeter -n -t test.jmx -l results.jtl

Results Issues

Missing Metrics

Symptoms:
- Incomplete dashboard
- Missing percentiles

Solution:

bash
 Verify JTL format includes all fields
 Use CSV format with full field list
lm run test.py --results-format csv --results-fields all

Check file wasn't truncated
wc -l results.jtl

Slow Report Generation

Symptoms:
- Report takes > 5 minutes
- Timeout during export

Solution:

python
 Generate report in background

```
report = lm.reports.create(
    execution_id=execution_id,
    async=True
)

Poll for completion
while report.status == "processing":
    time.sleep(30)
    report = lm.reports.get(report.id)

Download when ready
lm.reports.download(report.id, output="report.html")
```

Integration Issues

CI/CD Pipeline Failures

Symptoms:
- Tests pass locally but fail in CI
- Different results between runs

Troubleshooting:

```
yaml
 Ensure consistent environment
 .loadmagic-ci.yaml
version: 1

environment:
  loadmagic_version: "latest"

execution:
```

```
  users: "${USERS:-100}"
  spawn_rate: "${SPAWN_RATE:-10}"
  duration: "${DURATION:-5m}"

results:
  threshold:
    error_rate: 1%
    avg_response: 500ms
```

Webhook Not Firing

Symptoms:
- No webhook notification after test
- 404 on webhook endpoint

Solution:

```
python
 Verify webhook URL
import requests

Test endpoint exists
response = requests.get(webhook_url, timeout=5)
print(f"Status: {response.status_code}")

Check LoadMagic webhook config
lm.webhooks.list()
```

Performance Issues

Slow Script Execution

Symptoms:

- Throughput much lower than expected
- Long ramp-up times

Optimization Tips:

python
 1. Reduce think time variance
class OptimizedUser(HttpUser):
    wait_time = constant(1) More consistent

 2. Use HTTP keep-alive
class OptimizedUser(HttpUser):
     In host configuration, enable keep-alive
     Most efficient for same-host requests

 3. Disable SSL verification if needed (dev only!)
class OptimizedUser(HttpUser):
     Not recommended for production
    pass

4. Run distributed
 Use multiple JMeter/Locust workers
lm run test.py --workers 4

Agent Issues

AI Agent Not Responding

Symptoms:
- Carrie/George shows "unavailable"
- Request timeouts

Solution:

bash
 Check agent status
lm agents status

Restart agent
lm agents restart carrie

Or use different agent
lm agents.rupert.ask("How to extract JSON?")

Poor Agent Responses

Symptoms:
- Incomplete or incorrect suggestions
- Wrong regex patterns

Solution:

python
 Provide more context
response = lm.agents.rupert.generate_regex(
    sample_data='{"token": "abc123", "user": {"id": 456}}',
    extract_field="token",
    context="json",   More specific
    examples=[Provide examples
        {"input": '{"token": "xyz"}', "output": "xyz"},
        {"input": '{"token": "abc"}', "output": "abc"}
    ]
)

Getting Help

Debug Mode

bash
 Enable verbose logging
export LOADMAGIC_DEBUG=true
lm run test.py --verbose 2>&1 | tee debug.log

Collect Diagnostics

bash
 Generate diagnostic report
lm diagnostics collect --output diagnostics.zip

Include:
 - Version info
 - Config
 - Recent logs
 - Test results

Support Contact

Channel	Response Time
Email: support@loadmagic.ai	24-48 hours
In-app chat	1-4 hours
Enterprise support	< 1 hour

Summary Checklist

- [ ] API key valid and configured
- [ ] HAR file properly formatted

- [ ] Target accessible from execution environment
- [ ] Sufficient resources (CPU, memory, network)
- [ ] Correct permissions and authentication
- [ ] Results exported and stored

Next: [Core Usage Index](../4-Core-Usage/data-connectors.md)

4. Core Usage

Data Connectors

Overview

Data connectors in LoadMagic.AI enable you to import test data from various sources and formats. This guide covers all supported data connectors and their configuration.

Supported Data Sources

1. HAR Files (HTTP Archive)

The primary data source for LoadMagic.AI.

Supported Formats

Format	Extension	Support Level
Standard HAR	.har	Full
HAR with content	.har	Full
Chrome DevTools	.har	Full
Firefox Network Export	.har	Full

| Charles Proxy | .har | Partial |

Configuration

python
 Python SDK - HAR upload
from loadmagic import LoadMagic

lm = LoadMagic(api_key="your_key")

Upload and process
har = lm.hars.upload(
    file_path="./recordings/user-flow.har",
    options={
        "filter_static": True,   Remove CSS/JS/images
        "filter_domains": [Include only specific domains
            "api.example.com",
            "app.example.com"
        ],
        "exclude_domains": [Exclude domains
            "analytics.example.com",
            "cdn.example.com"
        ],
        "extract_auth": True,   Auto-detect auth patterns
        "group_by_url": True   Group similar requests
    }
)

print(f"Processed {har.endpoint_count} endpoints")
print(f"Auth patterns: {har.auth_patterns}")

Filtering Options

yaml

```
 har-filter-config.yaml
filters:
  By file type
 static_resources:
  - ".css"
  - ".js"
  - ".png"
  - ".jpg"
  - ".woff2"
  - ".svg"

  By domain
 include_domains:
  - api.example.com
  - app.example.com

 exclude_domains:
  - analytics.
  - tracking.

  By HTTP method
 methods:
  - GET
  - POST
  - PUT
  - DELETE

  By response code
 status_codes:
  exclude: [301, 302, 304]   Redirects
```

```
  By URL pattern
 url_patterns:
   include:
     - "/api/"
     - "/app/"
   exclude:
     - "/health"
     - "/metrics"
```

---

2. CSV Data Files

Import test data from CSV files.

CSV Format Requirements

```
csv
username,password,email,user_id
user1,pass123,user1@example.com,1001
user2,pass456,user2@example.com,1002
user3,pass789,user3@example.com,1003
```

Configuration

```
python
 Python SDK - CSV data import
from loadmagic import LoadMagic

lm = LoadMagic(api_key="your_key")

Import CSV
data_source = lm.data.import_csv(
    file_path="./test-data/users.csv",
```

```
    options={
        "delimiter": ",",           Default comma
        "quote_char": "\"",         Default double quote
        "has_header": True,         First row is header
        "encoding": "utf-8",        File encoding
        "skip_rows": 0,             Rows to skip
        "columns": [                Column mapping
            {"source": "username", "target": "username"},
            {"source": "password", "target": "password"},
            {"source": "email", "target": "email"}
        ]
    }
)

Use in script
class CSVUser(HttpUser):
    def on_start(self):
         Get data for this virtual user
        data = self.environment.data["users"][self.user_id % len(self.environment.data["users"])]
        self.username = data["username"]
        self.password = data["password"]
```

---

3. JSON Data Files

Import structured test data from JSON.

JSON Format

```
json
[
  {
```

```
    "username": "user1",
    "password": "pass123",
    "profile": {
      "firstName": "John",
      "lastName": "Doe"
    }
  },
  {
    "username": "user2",
    "password": "pass456",
    "profile": {
      "firstName": "Jane",
      "lastName": "Smith"
    }
  }
]
```

Configuration

```
python
 JSON data import
data_source = lm.data.import_json(
    file_path="./test-data/test-users.json",
    options={
        "array_path": "$.users",      JSONPath to data array
        "encoding": "utf-8"
    }
)
```

---

4. Database Connectors

Connect directly to databases for test data.

Supported Databases

Database	Support Level
PostgreSQL	Full
MySQL	Full
MongoDB	Full
SQL Server	Beta

PostgreSQL Example

```
python
 Database connector - PostgreSQL
from loadmagic.connectors import DatabaseConnector

db = DatabaseConnector(
    type="postgresql",
    host="db.example.com",
    port=5432,
    database="testdb",
    username="testuser",
    password="testpass"
)

Query test data
users = db.query("""
    SELECT id, username, email
    FROM users
    WHERE status = 'active'
    LIMIT 1000
""")
```

Use in script

```
class DBUser(HttpUser):
    def on_start(self):
        user_data = self.environment.db_data["users"][self.user_id]
        self.user_id = user_data["id"]
```

---

5. REST API Connector

Import data from existing APIs.

Configuration

python

```
 API connector
api = lm.connectors.api(
    base_url="https://api.example.com",
    auth={
        "type": "bearer",
        "token": "your_token"
    },
    headers={
        "X-Custom-Header": "value"
    }
)

Fetch data
products = api.get("/products", params={"limit": 100})
```

Use in script

```
class APIUser(HttpUser):
    def on_start(self):
        products = self.environment.api_data["products"]
```

```
        self.product = products[self.user_id % len(products)]
```

Data Transformation

Transform Functions

```
python
 Data transformation pipeline
from loadmagic.transforms import TransformPipeline

pipeline = TransformPipeline([
     Rename fields
    Transform.rename("user_id", "id"),

     Filter records
    Transform.filter(lambda r: r["status"] == "active"),

     Add computed fields
    Transform.add_field("full_name", lambda r: f"{r['first_name']} {r['last_name']}"),

     Hash sensitive data
    Transform.hash_field("password", algorithm="sha256"),

     Generate UUIDs
    Transform.add_field("session_id", lambda: uuid.uuid4().hex),

     Normalize dates
    Transform.normalize_dates("created_at", format="%Y-%m-%d")
])
```

Apply transformation

transformed_data = pipeline.apply(original_data)

Environment-Specific Data

Multi-Environment Support

python
 Environment-specific data
from loadmagic.environments import EnvironmentConfig

env = EnvironmentConfig(
    default="dev",
    environments={
        "dev": {
            "api_url": "https://dev-api.example.com",
            "data_file": "./data/dev-users.json"
        },
        "staging": {
            "api_url": "https://staging-api.example.com",
            "data_file": "./data/staging-users.json"
        },
        "prod": {
            "api_url": "https://api.example.com",
            "data_file": "./data/prod-users.json"
        }
    }
)

Use environment
script = lm.scripts.generate(
    har_file="./recording.har",

```
    environment=env.get_current()   Or specify: env.get("staging")
)
```

Data Security

Sensitive Data Handling

```
python
 Mark sensitive fields
data_source = lm.data.import_csv(
    file_path="./sensitive-data.csv",
    sensitive_fields=[
        "password",
        "ssn",
        "credit_card",
        "api_key"
    ],
     Options:
    mask_in_logs: True,       Mask in test output
    encrypt_at_rest: True,    Encrypt stored data
    exclude_from_export: True   Don't include in exports
)
```

Summary

Connector	Use Case	Complexity
HAR Files	Browser recordings	Low
CSV	User credentials, test data	Low
JSON	Complex structured data	Low

| Database | Dynamic data from production | Medium |
| REST API | External service data | Medium |

Next: [Model Integration](./model-integration.md)

4.1 Model Integration

Model Integration

Overview

LoadMagic.AI leverages AI models through specialized agents to automate performance testing tasks. This guide covers how to integrate and customize these AI capabilities.

AI Agents Overview

LoadMagic.AI provides five specialized AI agents, each with unique capabilities:

Agent	Primary Function	Model Used
George	JMeter expertise, debugging	Claude/GPT-4
Rupert	Regex generation	Custom regex model
Suzy	Script generation	GPT-4
Carrie	Correlation automation	Claude
Quinn	QA assessment	Claude

Agent Configuration

Default Configuration

python
 Default agent configuration
from loadmagic.agents import AgentConfig

config = AgentConfig(
     All agents
    default_model="claude-3-sonnet",
    temperature=0.7,
    max_tokens=4000,

     Per-agent overrides
    agent_settings={
        "george": {
            "model": "claude-3-sonnet",
            "specialization": "jmeter"
        },
        "rupert": {
            "model": "gpt-4",
            "regex_optimization": True
        },
        "suzy": {
            "model": "gpt-4",
            "code_style": "pythonic"
        }
    }
)

Initialize with config
lm = LoadMagic(api_key="your_key", agent_config=config)

Custom Model Settings

python
 Customize model parameters per agent
from loadmagic.agents import George, Rupert, Suzy, Carrie, Quinn

George - More verbose for debugging
george = George(
    model="claude-3-opus",
    temperature=0.9,
    max_tokens=8000,
    include_examples=True
)

Rupert - Precise regex generation
rupert = Rupert(
    model="gpt-4-turbo",
    temperature=0.2,   Low for precision
    max_tokens=2000,
    optimize_for="precision"
)

Suzy - Creative script generation
suzy = Suzy(
    model="gpt-4",
    temperature=0.7,
    max_tokens=4000,
    code_style="modern"
)

Using AI Agents

George - JMeter Expert

```
python
 Ask questions about JMeter
from loadmagic.agents import George

george = George(api_key="your_key")

General question
response = george.ask(
    "How do I configure SSL keystore in JMeter?"
)
print(response.answer)

Specific technical question
response = george.ask(
    "Why am I getting 'Connection timed out' errors in my HTTP requests?",
    context={
        "test_type": "load",
        "protocol": "https",
        "target": "api.example.com"
    }
)
print(response.answer)
print(response.fix_steps)

Debug error
response = george.debug(
    error_message="Response message: Bad request",
    sampler_type="HTTP Request",
    response_data={"error": "Invalid token"}
)
print(response.root_cause)
print(response.suggested_fix)
```

Rupert - Regex Expert

python
 Generate regex patterns
from loadmagic.agents import Rupert

rupert = Rupert(api_key="your_key")

JSON extraction
response = rupert.generate_regex(
    sample_data='{"user_id": 12345, "token": "abc123xyz"}',
    extract_field="token",
    context="json",
    output_format="jmeter"   or "locust"
)
print(response.regex)   "token":\s"([^"]+)"

Complex pattern
response = rupert.generate_regex(
    sample_data='Session: abc123; Expires: 2026-02-23T15:30:00Z;',
    extract_field="session_id",
    context="text",
    boundary_matching=True
)
print(response.regex)

HTML extraction
response = rupert.generate_regex(
    sample_data='<input name="csrf_token" value="xyz789" />',
    extract_field="csrf_token",
    context="html"
)
print(response.regex)

Suzy - Script Generator

```
python
 Generate scripts from description
from loadmagic.agents import Suzy

suzy = Suzy(api_key="your_key")

Generate JMeter script
script = suzy.generate(
    description="Create a test that:
    1. Logs in with username/password
    2. Gets user profile
    3. Updates user preferences
    4. Logs out",
    framework="jmeter",
    language="groovy",
    options={
        "correlation": "auto",
        "assertions": True,
        "think_time": True
    }
)
print(script.content)

Convert script language
converted = suzy.convert(
    script=jmeter_script,
    from_language="groovy",
    to_language="python"
)
print(converted.content)
```

```
Fix script errors
fixed = suzy.fix(
    script=broken_script,
    error="SyntaxError: unexpected indent",
    framework="locust"
)
print(fixed.content)

Carrie - The Correlator

python
 Automated correlation
from loadmagic.agents import Carrie

carrie = Carrie(api_key="your_key")

Full correlation pipeline
result = carrie.correlate(
    script=generated_script,
    responses=sample_responses,
    mode="agent",   passive, suggest, or agent
    options={
        "auto_repair": True,
        "validation": True,
        "track_dependencies": True
    }
)

print(f"Correlations found: {len(result.correlations)}")
print(f"Extractors added: {len(result.extractors)}")

Each correlation
for corr in result.correlations:
```

```
    print(f"  {corr.name}: {corr.source}  {corr.target}")
```

Quinn - QA Assessor

```
python
 Quality assessment
from loadmagic.agents import Quinn

quinn = Quinn(api_key="your_key")

Analyze script
assessment = quinn.assess(
    script=test_script,
    check_types=[
        "assertions",
        "correlations",
        "configuration",
        "best_practices"
    ],
    auto_fix=True,
    strictness="high"
)

print(f"Quality Score: {assessment.score}/100")
print(f"Issues Found: {len(assessment.issues)}")

Issues breakdown
for issue in assessment.issues:
    print(f"  [{issue.severity}] {issue.type}: {issue.message}")
    if issue.fix:
        print(f"    Fix: {issue.fix.description}")
```

Custom Agent Training

Fine-tuning for Enterprise

```
python
 Fine-tune agent on your patterns
from loadmagic.agents import AgentTrainer

trainer = AgentTrainer(
    agent="rupert",
    api_key="your_key"
)

Add custom examples
trainer.add_examples([
    {
        "input": "Extract user ID from response: {\"data\":{\"user\":{\"id\":123}}}",
        "output": "'data'.'user'.'id'",
        "context": "json_path"
    },
    {
        "input": "Extract token from: Authorization: Bearer eyJhbGciOiJIUzI1NiJ9",
        "output": "Bearer\\s+(\\S+)",
        "context": "regex"
    }
])

Train model
training_job = trainer.train(
    epochs=10,
    validation_split=0.2
)
```

print(f"Training status: {training_job.status}")
print(f"Model version: {training_job.model_version}")

Prompt Engineering

Custom Prompts

python
 Customize agent prompts
from loadmagic.agents import PromptTemplate

Custom George prompt
custom_george = George(
    prompt_template=PromptTemplate("""
        You are a JMeter expert specializing in:
        - REST API testing
        - Authentication flows
        - Performance optimization

        Context: {context}
        Question: {question}

        Provide a detailed answer with:
        1. Explanation
        2. Example configuration
        3. Best practices
    """)
)

Custom Suzy prompt
custom_suzy = Suzy(

```
    prompt_template=PromptTemplate("""
        Generate {framework} performance test scripts.

        Requirements:
        {requirements}

        Follow these coding standards:
        - Use type hints
        - Include docstrings
        - Handle errors gracefully

        Framework: {framework}
        Language: {language}
    """)
)
```

Model Versioning

Version Management

python
 Manage model versions
from loadmagic.models import ModelManager

manager = ModelManager(api_key="your_key")

List available models
models = manager.list_models(agent="george")
for model in models:
    print(f"{model.name} (v{model.version}) - {model.status}")

Use specific version

```
george = George(
    model="claude-3-sonnet",
    version="2024-01"
)

A/B testing with different models
experiment = {
    "control": {"agent": "george", "model": "claude-3-haiku"},
    "variant": {"agent": "george", "model": "claude-3-sonnet"}
}

results = manager.run_experiment(
    experiment=experiment,
    test_cases=validation_set,
    metrics=["accuracy", "speed", "cost"]
)
```

Cost Optimization

Token Usage Tracking

```
python
 Track and optimize usage
from loadmagic.analytics import UsageTracker

tracker = UsageTracker(api_key="your_key")

Get usage report
usage = tracker.get_usage(
    period="month",
    breakdown_by="agent"
)
```

```
print(f"Total tokens: {usage.total_tokens}")
print(f"Cost: ${usage.total_cost}")

Optimize suggestions
suggestions = tracker.optimize()
for suggestion in suggestions:
    print(f"  {suggestion.title}")
    print(f"    Potential savings: {suggestion.savings}")
```

Summary

Agent	Best For	Customization
George	Debugging, knowledge	Prompt templates
Rupert	Regex patterns	Examples, precision
Suzy	Script generation	Code style, frameworks
Carrie	Correlation	Validation rules
Quinn	Quality checks	Check types, severity

Next: [Pipeline Orchestration](./pipeline-orchestration.md)

4.2 Pipeline Orchestration

Pipeline Orchestration

Overview

Pipeline orchestration in LoadMagic.AI enables you to create complex, multi-step testing workflows that can include data preparation, test execution, validation, and reporting.

Basic Pipeline Concepts

Pipeline Structure

python
 Pipeline definition
from loadmagic.pipelines import Pipeline, Stage

pipeline = Pipeline(
    name="api-performance-pipeline",
    stages=[
        Stage(name="prepare",
              action=prepare_data),
        Stage(name="baseline",
              action=run_baseline_test),
        Stage(name="load",
              action=run_load_test),
        Stage(name="validate",
              action=validate_results),
        Stage(name="report",
              action=generate_report)
    ]
)

Pipeline Builder

Visual Pipeline Builder

yaml
 pipeline.yaml

```
name: "E2E Performance Pipeline"

stages:
  - name: "Data Preparation"
    type: "transform"
    config:
      input: "./data/test-users.json"
      transform: "normalize"
      output: "${WORKSPACE}/prepared-data.json"

  - name: "Generate Scripts"
    type: "script-generation"
    config:
      har_file: "${WORKSPACE}/recording.har"
      framework: "locust"
      correlation: "auto"

  - name: "Baseline Test"
    type: "execution"
    config:
      script: "${WORKSPACE}/api-test.py"
      users: 10
      duration: "2m"
      environment: "staging"

  - name: "Load Test"
    type: "execution"
    config:
      script: "${WORKSPACE}/api-test.py"
      users: 1000
      duration: "30m"
```

```
      environment: "staging"

  - name: "Validation"
    type: "validation"
    config:
      thresholds:
        error_rate: "< 1%"
        avg_response: "< 500ms"
        p95_response: "< 1000ms"

  - name: "Report"
    type: "notification"
    config:
      channels:
        - email: "team@example.com"
        - slack: "performance"
```

Python Pipeline Definition

```
python
 Python pipeline definition
from loadmagic.pipelines import Pipeline, Stage, Step

Define the pipeline
pipeline = Pipeline(
    name="api-performance-pipeline",
    description="Complete API performance testing pipeline",
    version="1.0"
)

Stage 1: Data Preparation
prepare_stage = Stage(
    name="prepare",
```

```
    steps=[
        Step(
            action="load_data",
            params={
                "source": "csv",
                "path": "./data/users.csv"
            }
        ),
        Step(
            action="transform",
            params={
                "operations": ["normalize", "hash_passwords"]
            }
        )
    ]
)

Stage 2: Test Execution
test_stage = Stage(
    name="execute",
    steps=[
        Step(
            action="execute_test",
            params={
                "script": "./scripts/api-test.py",
                "users": 100,
                "duration": "10m"
            }
        )
    ]
)
```

Stage 3: Validation
validate_stage = Stage(
    name="validate",
    steps=[
        Step(
            action="check_thresholds",
            params={
                "avg_response_time": 500,
                "error_rate": 1.0
            }
)
    ]
)

Add stages to pipeline
pipeline.add_stages([prepare_stage, test_stage, validate_stage])

Conditional Execution

Branching Logic

python
from loadmagic.pipelines import ConditionalStage

Conditional execution based on results
pipeline = Pipeline(
    name="smart-pipeline",
    stages=[
        Stage(name="baseline", action=run_baseline),

         Only run full test if baseline passes

```
        ConditionalStage(
            name="full-test",
            condition=lambda ctx: ctx.baseline.error_rate < 5,
            if_true=Stage(name="load-test", action=run_load),
            if_false=Stage(name="alert", action=send_alert)
        ),

         Conditional cleanup
        ConditionalStage(
            name="cleanup",
            condition=lambda ctx: ctx.config.get("cleanup", True),
            if_true=Stage(name="clean", action=cleanup)
        )
    ]
)
```

Error Handling

```
python
 Error handling and recovery
from loadmagic.pipelines import Stage, ErrorPolicy

pipeline = Pipeline(
    name="resilient-pipeline",
    error_policy=ErrorPolicy(
        retry_count=3,
        retry_delay=30,   seconds
        backoff="exponential",
        on_failure="continue",   or "abort", "cleanup"
        fallback_action=run_fallback_test
    ),
    stages=[
```

```
        Stage(name="test", action=run_primary_test),
        Stage(name="backup-test", action=run_backup_test)
    ]
)
```

Parallel Execution

Concurrent Stages

python

```
from loadmagic.pipelines import ParallelStage, Pipeline

Run multiple tests in parallel
pipeline = Pipeline(
    name="parallel-pipeline",
    stages=[
         Sequential: Setup
        Stage(name="setup", action=setup),

         Parallel: Run tests on different endpoints
        ParallelStage(
            name="parallel-tests",
            stages=[
                Stage(name="users-api", action=test_users_api),
                Stage(name="products-api", action=test_products_api),
                Stage(name="orders-api", action=test_orders_api)
            ],
            max_concurrent=3
        ),

         Sequential: Aggregate results
```

```
        Stage(name="aggregate", action=aggregate_results)
    ]
)
```

Pipeline Triggers

Event-Based Triggers

yaml
 trigger-config.yaml
triggers:
  - name: "on-merge"
    type: "webhook"
    events: ["pull_request.merged"]
    action: pipeline.run
    config:
      pipeline: "full-test-pipeline"
      branch: "main"

  - name: "scheduled-nightly"
    type: "schedule"
    cron: "0 2   "   2 AM daily
    action: pipeline.run
    config:
      pipeline: "nightly-pipeline"
      parameters:
        users: 500
        duration: "1h"

  - name: "on-deployment"
    type: "webhook"

```
    events: ["deployment.completed"]
    action: pipeline.run
    config:
      pipeline: "smoke-test-pipeline"

---

Context & Variables

Pipeline Context

python
 Accessing pipeline context
from loadmagic.pipelines import PipelineContext

def my_stage(context: PipelineContext):
     Access previous stage results
    baseline_results = context.get_stage("baseline").results

     Access configuration
    users = context.config.get("users", 100)

     Access environment variables
    api_url = context.env["API_URL"]

     Set variables for later stages
    context.set_var("baseline_passed", baseline_results.error_rate < 1)

     Access shared data
    shared_data = context.shared["test_data"]

    return {"status": "success", "output": baseline_results}
```

Variable Types

yaml
 Variables in pipeline
variables:
   Static variables
  API_URL: "https://api.example.com"
  THRESHOLD_ERROR_RATE: "1"

   Dynamic from context
  BASELINE_RESULTS: "${stages.baseline.results.error_rate}"

   Computed
  USER_MULTIPLIER: "${env.USERS_MULTIPLIER || 1}"

   Secrets (from secure vault)
  API_KEY: "${vault:api-key}"

Complete Pipeline Example

python
 Complete pipeline example
from loadmagic.pipelines import Pipeline, Stage, Step, ParallelStage
from loadmagic.notifications import EmailNotifier, SlackNotifier

Define pipeline
pipeline = Pipeline(
    name="api-performance-pipeline",
    description="Complete API performance testing workflow",

     Configuration

```
    config={
        "default_users": 100,
        "default_duration": "10m",
        "timeout": 3600
    },

     Notifications
    notifications=[
        EmailNotifier(
            recipients=["team@example.com"],
            on=["start", "success", "failure"]
        ),
        SlackNotifier(
            channel="performance",
            on=["failure"]
        )
    ]
)

Add stages
pipeline.add_stage(Stage(
    name="prepare",
    description="Prepare test data",
    steps=[
        Step(action="load_csv", params={"path": "./data/users.csv"}),
        Step(action="transform", params={"operations": ["normalize"]})
    ]
))

pipeline.add_stage(Stage(
    name="baseline",
    description="Run baseline test",
```

```
    steps=[
        Step(action="execute", params={
            "script": "./scripts/api.py",
            "users": 10,
            "duration": "2m"
        })
    ]
))

pipeline.add_stage(Stage(
    name="load-test",
    description="Run full load test",
    condition=lambda ctx: ctx.baseline.error_rate < 5,
    steps=[
        Step(action="execute", params={
            "script": "./scripts/api.py",
            "users": 1000,
            "duration": "30m"
        })
    ]
))

pipeline.add_stage(Stage(
    name="validate",
    description="Validate results against thresholds",
    steps=[
        Step(action="check_thresholds", params={
            "avg_response_time": 500,
            "p95_response_time": 1000,
            "error_rate": 1.0
        })
    ]
```

```
))

pipeline.add_stage(Stage(
    name="report",
    description="Generate and send reports",
    steps=[
        Step(action="generate_html"),
        Step(action="send_notifications")
    ]
))

Execute pipeline
result = pipeline.execute(
    config={
        "users": 500,
        "environment": "staging"
    }
)

print(f"Pipeline status: {result.status}")
print(f"Duration: {result.duration}s")
```

Summary

Feature	Use Case	Complexity
Sequential stages	Simple workflows	Low
Parallel execution	Concurrent testing	Medium
Conditional logic	Decision-based flows	Medium
Error handling	Resilient pipelines	Medium
Triggers	Automation	Low

| Variables | Dynamic configuration | Low |

Next: [Scheduling & Retries](./scheduling-retries.md)

4.3 Scheduling & Retries

Scheduling & Retries

Overview

LoadMagic.AI provides robust scheduling capabilities for automated test execution and retry mechanisms for handling transient failures.

Scheduled Execution

Cron-Based Scheduling

python
 Schedule tests using cron expressions
from loadmagic.scheduling import Schedule, CronExpression

Daily schedule
daily = Schedule(
    name="daily-load-test",
    cron="0 2   ",   2 AM daily
    timezone="UTC",
    enabled=True
)

Weekly schedule (Monday at 3 AM)

```
weekly = Schedule(
    name="weekly-full-test",
    cron="0 3   1",   3 AM Monday
    timezone="America/New_York",
    enabled=True
)

Every 15 minutes
frequent = Schedule(
    name="smoke-test",
    cron="/15    ",
    enabled=True
)

Custom interval (every 6 hours)
interval = Schedule(
    name="periodic-check",
    interval_hours=6,
    enabled=True
)

Schedule Configuration

yaml
 schedule-config.yaml
schedules:
  - name: "nightly-regression"
    enabled: true
    cron: "0 2   "
    timezone: "UTC"

     What to run
    pipeline: "full-regression-pipeline"
```

```
    Parameters
    parameters:
      users: 500
      duration: "1h"
      environment: "staging"

    Notifications
    notify:
      on_start: false
      on_success: true
      on_failure: true

    Retry config
    retry:
      enabled: true
      max_attempts: 2
      delay_minutes: 30

  - name: "pre-deployment-check"
    enabled: true
    cron: "0 /2   "   Every 2 hours
    pipeline: "smoke-test"
```

Retry Mechanisms

Automatic Retries

python

```
 Configure retry behavior
from loadmagic.execution import ExecutionConfig, RetryPolicy
```

```
config = ExecutionConfig(
    script="./test.py",

     Retry policy
    retry=RetryPolicy(
        enabled=True,
        max_attempts=3,
        strategy="exponential",   or "fixed", "linear"
        initial_delay=30,   seconds
        max_delay=300,   5 minutes max
        backoff_multiplier=2,

         What to retry on
        retry_on_errors=[
            "connection_timeout",
            "503_service_unavailable",
            "504_gateway_timeout"
        ],

         Don't retry on
        dont_retry_on_errors=[
            "400_bad_request",
            "401_unauthorized",
            "404_not_found"
        ]
    )
)

Execute with retry
result = lm.executions.start(config)

Retry Strategies
```

python
from loadmagic.execution import RetryStrategy

Fixed delay
fixed = RetryStrategy.fixed(
    attempts=3,
    delay_seconds=30
)

Exponential backoff
exponential = RetryStrategy.exponential(
    attempts=5,
    initial_delay=10,
    multiplier=2,
    max_delay=120
)

Linear backoff
linear = RetryStrategy.linear(
    attempts=4,
    delay_seconds=15
)

Custom
custom = RetryStrategy(
    attempts=3,
    delays=[10, 30, 60]   Specific delays
)

Advanced Scheduling

Event-Driven Scheduling

python
 Trigger on external events
from loadmagic.scheduling import EventTrigger

GitHub webhook trigger
trigger = EventTrigger(
    provider="github",
    events=["push", "pull_request"],
    branches=["main", "develop"],
    paths=["api/", "services/"],
    action=lambda event: pipeline.run(
        parameters={
            "commit": event.commit_id,
            "branch": event.branch
        }
)
)

Deploy pipeline trigger
deploy_trigger = EventTrigger(
    provider="custom",
    endpoint="/webhooks/deploy",
    action=lambda payload: pipeline.run(
        parameters={
            "environment": payload.environment,
            "version": payload.version
        }
)
)

Dependency Scheduling

python

```
 Run after other pipelines complete
from loadmagic.scheduling import DependencyTrigger

trigger = DependencyTrigger(
    pipeline="build-pipeline",
    events=["completed"],
    condition=lambda result: result.status == "success",
    action=lambda: pipeline.run()
)

Run after multiple pipelines
multi_trigger = DependencyTrigger(
    pipelines=["build-pipeline", "security-scan"],
    condition="all_success",   or "any_success"
    action=lambda: pipeline.run()
)
```

Execution Queues

Queue Management

```
python
from loadmagic.execution import Queue, Priority

Define execution queue
queue = Queue(
    name="performance-tests",
    priority=Priority.NORMAL,

     Queue settings
    max_concurrent=5,
    max_queue_size=20,
```

```
    Priority rules
    priority_rules={
        "production": Priority.HIGH,
        "staging": Priority.NORMAL,
        "development": Priority.LOW
    }
)

Enqueue execution
execution_id = lm.executions.enqueue(
    script="./test.py",
    queue=queue,
    priority=Priority.HIGH
)

Queue status
status = lm.queues.get_status(queue.name)
print(f"Position: {status.position}")
print(f"Wait time: {status.estimated_wait}")

---

Best Practices

Scheduling Guidelines

yaml
 Recommended schedules
guidelines:
   Don't run heavy tests during peak hours
  avoid_during:
    - "9:00-11:00"   Business hours
```

```
    - "14:00-16:00"  Business hours

  Recommended times
  recommended:
    - "2:00"   Night (UTC)
    - "6:00"   Early morning (UTC)

  Stagger different test types
  stagger:
    smoke_test: "/15    "   Every 15 min
    regression: "0 3   "       3 AM daily
    full_load: "0 2   0"       2 AM Sunday

  Resource allocation
  resource_limits:
    concurrent_tests: 5
    max_users_per_test: 10000
```

Monitoring Schedules

Track Execution Health

```
python
from loadmagic.analytics import ScheduleAnalytics

analytics = ScheduleAnalytics()

Get schedule performance
performance = analytics.get_schedule_performance(
    schedule_id="nightly-regression",
    period="30d"
```

```
)

print(f"Total runs: {performance.total_runs}")
print(f"Success rate: {performance.success_rate}%")
print(f"Avg duration: {performance.avg_duration}min")
print(f"Missed runs: {performance.missed_runs}")

Get failure analysis
failures = analytics.get_failure_analysis(
    period="7d"
)
for failure in failures:
    print(f"{failure.schedule}: {failure.count} failures")
    print(f"  Reason: {failure.root_cause}")
```

Summary

Feature	Description	Best For
Cron scheduling	Time-based execution	Regular tests
Event triggers	On-change execution	CI/CD integration
Dependency triggers	Pipeline chains	Complex workflows
Retry policies	Failure recovery	Unstable environments
Queues	Resource management	Team environments

Next: [Monitoring & Observability](./monitoring-observability.md)

4.4 Monitoring & Observability

Monitoring & Observability

Overview

LoadMagic.AI provides comprehensive monitoring and observability features to track test execution, analyze results, and maintain system health.

Real-Time Monitoring

Live Execution Dashboard

python
 Monitor execution in real-time
from loadmagic.monitoring import ExecutionMonitor

monitor = ExecutionMonitor(execution_id="exec_abc123")

Stream real-time metrics
for metric in monitor.stream():
    print(f"[{metric.timestamp}]")
    print(f" Active users: {metric.active_users}")
    print(f" Requests/sec: {metric.rps}")
    print(f" Avg response: {metric.avg_response_time}ms")
    print(f" Error rate: {metric.error_rate}%")
    print(f" Errors: {metric.total_errors}")

Real-Time Metrics

Metric	Description	Alert Threshold
active_users	Current virtual users	> target
rps	Requests per second	< expected
avg_response_time	Mean response time	> 500ms

p95_response_time	95th percentile	> 1000ms
error_rate	Percentage of errors	> 1%
total_requests	Cumulative requests	N/A

Metrics Collection

Custom Metrics

python
 Define custom metrics
from loadmagic.metrics import Metric, Counter, Gauge, Histogram

Counter for request types
request_count = Counter(
    name="api_requests_total",
    description="Total API requests",
    labels=["endpoint", "method", "status"]
)

Gauge for active users
active_users = Gauge(
    name="loadtest_active_users",
    description="Current active users"
)

Histogram for response times
response_time = Histogram(
    name="api_response_time_seconds",
    description="API response time",
    buckets=[0.1, 0.25, 0.5, 1.0, 2.5, 5.0]
)

Use in script

```
class APIUser(HttpUser):
    @task
    def get_users(self):
        with response_time.time():
            response = self.client.get("/api/users")

        request_count.labels(
            endpoint="/api/users",
            method="GET",
            status=response.status_code
        ).inc()
```

Metrics Export

python

```
 Export metrics to external systems
from loadmagic.exporters import PrometheusExporter, DataDogExporter

Prometheus export
prometheus = PrometheusExporter(
    port=9090,
    path="/metrics"
)
prometheus.start()

DataDog export
datadog = DataDogExporter(
    api_key="your_datadog_key",
    tags=["env:staging", "team:platform"]
)
datadog.start()
```

Custom exporter
custom = CustomExporter(
    endpoint="https://metrics.example.com/push",
    batch_size=100,
    interval_seconds=10
)

Logging

Structured Logging

python
 Configure logging
from loadmagic.logging import Logger, LogLevel

logger = Logger(
    name="loadtest",
    level=LogLevel.DEBUG,
    format="json",
    output=["file", "stdout"]
)

Log execution events
logger.info("Test starting",
    users=1000,
    duration="30m",
    target="api.example.com"
)

logger.debug("Request sent",
    endpoint="/api/users",
    method="POST",

```
    request_id="req_abc123"
)

logger.warning("High response time",
    endpoint="/api/search",
    response_time_ms=2500,
    threshold_ms=1000
)

logger.error("Request failed",
    endpoint="/api/checkout",
    error="connection_timeout",
    attempt=3
)
```

Log Aggregation

```
yaml
 log-config.yaml
logging:
  level: INFO
  format: json

  outputs:
    - type: stdout
    - type: file
      path: /var/log/loadmagic/test.log
      rotation:
        max_size: 100MB
        max_files: 10
    - type: syslog
      host: syslog.example.com
      port: 514
```

```
    - type: elasticsearch
      hosts:
        - elasticsearch:9200
      index: loadmagic-logs

  filters:
    - type: exclude
      pattern: "/health"
    - type: include
      min_level: WARNING

---

Dashboards

Pre-built Dashboards

python
 Access pre-built dashboards
from loadmagic.dashboards import Dashboard

Execution dashboard
exec_dashboard = Dashboard.get("execution")
exec_dashboard.configure(
    refresh_interval=5,
    time_range="last_1h"
)

Results dashboard
results_dashboard = Dashboard.get("results")
results_dashboard.configure(
    metrics=["avg_response", "p95", "p99", "error_rate"],
    group_by="test_name"
```

```
)
```

Custom Dashboards

```
python
 Create custom dashboard
from loadmagic.dashboards import Dashboard, Panel, Chart

dashboard = Dashboard(
    name="Performance Overview",
    description="Custom performance monitoring dashboard"
)

Add panels
dashboard.add_panel(Panel(
    title="Response Time Trend",
    chart=Chart(
        type="line",
        metrics=["avg_response_time", "p95_response_time", "p99_response_time"],
        time_range="24h",
        group_by="test_name"
    )
))

dashboard.add_panel(Panel(
    title="Error Rate by Endpoint",
    chart=Chart(
        type="bar",
        metrics=["error_rate"],
        group_by="endpoint"
    )
))
```

Save and share
dashboard.save()
dashboard.share(team="platform")

Alerting

Alert Configuration

python
 Configure alerts
from loadmagic.alerts import AlertRule, NotificationChannel

Email notification
email = NotificationChannel(
    type="email",
    recipients=["team@example.com"],
    template="alert-template.html"
)

Slack notification
slack = NotificationChannel(
    type="slack",
    webhook_url="https://hooks.slack.com/...",
    channel="alerts"
)

Alert rules
alert = AlertRule(
    name="High Error Rate",
    condition="error_rate > 5%",
    duration="5m",
    severity="critical",

```
    channels=[email, slack],
    message="Error rate exceeded 5% for 5 minutes"
)

alert2 = AlertRule(
    name="Slow Response",
    condition="p95_response_time > 2000",
    duration="10m",
    severity="warning",
    channels=[slack]
)
```

Alert Rules

```
yaml
 alert-rules.yaml
alerts:
  - name: "Critical Error Rate"
    condition: "error_rate > 5%"
    duration: "5m"
    severity: "critical"
    channels: ["email", "slack", "pagerduty"]

  - name: "High Response Time"
    condition: "avg_response_time > 1000"
    duration: "10m"
    severity: "warning"
    channels: ["slack"]

  - name: "P99 Threshold Breached"
    condition: "p99_response_time > 3000"
    duration: "5m"
```

```
    severity: "warning"
    channels: ["email"]

  - name: "Test Failed"
    condition: "execution_status == 'failed'"
    duration: "0m"
    severity: "critical"
    channels: ["email", "slack"]
```

Integration with External Systems

Prometheus Integration

```
yaml
 prometheus-config.yaml
scrape_configs:
  - job_name: 'loadmagic'
    static_configs:
      - targets: ['loadmagic-metrics:9090']
    metrics_path: '/metrics'
    scrape_interval: 15s
```

Grafana Integration

```
python
 Grafana datasource
datasource = {
    "name": "LoadMagic",
    "type": "prometheus",
    "url": "https://api.loadmagic.ai/prometheus",
    "access": "proxy",
```

```
    "basicAuth": True,
    "basicAuthUser": "your_user",
    "jsonData": {
        "timeInterval": "15s"
    }
}

Import dashboard
dashboard = lm.dashboards.export_grafana(dashboard_id="12345")
 Import into Grafana using Grafana API

Datadog Integration

python
 Datadog configuration
from loadmagic.integrations import Datadog

datadog = Datadog(
    api_key="your_api_key",
    app_key="your_app_key",
    tags=["env:production", "team:platform"]
)

Enable metrics export
datadog.enable_metrics(
    interval=10,   seconds
    metrics=["response_time", "error_rate", "throughput"]
)

Enable log export
datadog.enable_logs(
    index="loadmagic",
    service="loadtest"
```

)

Summary

Feature	Description	Integration
Real-time monitoring	Live execution tracking	Web UI, WebSocket
Custom metrics	Application-specific data	Prometheus, StatsD
Structured logging	Detailed event capture	ELK, Splunk
Dashboards	Visual analytics	Grafana, DataDog
Alerting	Proactive notifications	PagerDuty, Slack

Next: [Security Basics](./security-basics.md)

4.5 Security Basics

Security Basics

Overview

LoadMagic.AI provides comprehensive security features to protect your test data, API access, and execution environment. This guide covers essential security configurations.

Authentication

API Key Authentication

python

```
 Using API key
from loadmagic import LoadMagic

lm = LoadMagic(
    api_key="lm_api_xxxxxxxxxxxxx"
)

Verify authentication
status = lm.auth.verify()
print(f"Authenticated: {status.authenticated}")
print(f"User: {status.user_email}")
print(f"Plan: {status.plan}")

OAuth 2.0 / OIDC

python
 OAuth 2.0 authentication
from loadmagic.auth import OAuth

oauth = OAuth(
    client_id="your_client_id",
    client_secret="your_client_secret",
    authorization_url="https://loadmagic.auth.example.com/authorize",
    token_url="https://loadmagic.auth.example.com/token",
    scopes=["read", "write", "execute"]
)

Get authorization URL
auth_url = oauth.get_authorization_url(
    redirect_uri="https://yourapp.com/callback"
)

Exchange code for token
token = oauth.exchange_code(
```

```
    code="authorization_code",
    redirect_uri="https://yourapp.com/callback"
)

Initialize with OAuth token
lm = LoadMagic(token=token.access_token)
```

SSO / SAML (Enterprise)

```
yaml
 SAML configuration (Enterprise)
saml:
  enabled: true
  idp_entity_id: "https://idp.example.com"
  idp_sso_url: "https://idp.example.com/sso"
  idp_cert: "/path/to/idp-cert.pem"

  sp_entity_id: "loadmagic.yourcompany.com"
  sp_acs_url: "https://loadmagic.yourcompany.com/saml/acs"

  attribute_mapping:
    email: "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress"
    name: "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name"
    role: "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/role"

---

Authorization

Role-Based Access Control (RBAC)

yaml
 RBAC configuration
rbac:
```

```
roles:
  - name: "admin"
    description: "Full access"
    permissions:
      - "project:"
      - "script:"
      - "execution:"
      - "results:"
      - "settings:"
      - "users:"

  - name: "engineer"
    description: "Test execution and analysis"
    permissions:
      - "project:read"
      - "project:write"
      - "script:"
      - "execution:"
      - "results:read"

  - name: "viewer"
    description: "View-only access"
    permissions:
      - "project:read"
      - "results:read"

users:
  - email: "admin@company.com"
    role: "admin"

  - email: "engineer@company.com"
```

```
    role: "engineer"

  - email: "viewer@company.com"
    role: "viewer"
```

Permission Scopes

Scope	Description
project:read	View projects
project:write	Create/update projects
project:delete	Delete projects
script:read	View scripts
script:write	Create/update scripts
script:execute	Run tests
results:read	View results
results:export	Export results
settings:read	View settings
settings:write	Modify settings
users:manage	Manage users

Data Encryption

Encryption at Rest

```
python
 Encryption configuration
from loadmagic.security import EncryptionConfig

config = EncryptionConfig(
    algorithm="AES-256-GCM",
    key_rotation_days=90,
```

```
    Key management
    key_manager="aws-kms",   or "azure-keyvault", "gcp-kms"
    key_id="arn:aws:kms:us-east-1:123456789:key/xxx"
)

Mark sensitive fields
lm.data.mark_sensitive(
    fields=["password", "ssn", "credit_card"],
    encryption_required=True
)

Encryption in Transit

bash
 Force TLS 1.3
export LOADMAGIC_TLS_VERSION=1.3

Custom CA certificate
export LOADMAGIC_CA_CERT=/path/to/ca-bundle.crt

Verify connection
lm.health.check(verify_ssl=True)

---

Secret Management

Environment Variables

bash
 Secure environment setup
export LOADMAGIC_API_KEY="your_api_key"
export LOADMAGIC_SECRET_KEY="your_secret"
```

Use .env files (add to .gitignore)

```
.env
LOADMAGIC_API_KEY=your_key
LOADMAGIC_API_SECRET=your_secret
```

Vault Integration

python

```
HashiCorp Vault integration
from loadmagic.integrations import Vault

vault = Vault(
    url="https://vault.example.com",
    token="your_vault_token",
    namespace="admin"
)

Fetch secrets
secrets = vault.get_secrets(
    path="secret/data/loadmagic",
    keys=["api-key", "db-password"]
)

Use in configuration
config = {
    "api_key": secrets["api-key"],
    "db_password": secrets["db-password"]
}
```

Audit Logging

Enable Audit Logs

```
python
 Configure audit logging
from loadmagic.audit import AuditConfig

config = AuditConfig(
    enabled=True,
    log_level="INFO",

     What to log
    events=[
        "user.login",
        "user.logout",
        "project.create",
        "project.delete",
        "script.execute",
        "results.export",
        "settings.change"
    ],

     Where to send
    destinations=[
        {"type": "cloudwatch", "group": "loadmagic-audit"},
        {"type": "splunk", "url": "https://splunk.example.com"},
        {"type": "file", "path": "/var/log/loadmagic/audit.log"}
    ]
)
```

Audit Log Format

```
json
{
  "timestamp": "2026-02-23T10:30:00Z",
```

```
  "event": "script.execute",
  "user": {
    "id": "user_123",
    "email": "engineer@example.com",
    "ip": "192.168.1.100"
  },
  "resource": {
    "type": "script",
    "id": "script_abc",
    "name": "api-load-test"
  },
  "action": {
    "type": "execute",
    "parameters": {
      "users": 1000,
      "duration": "10m"
    }
  },
  "result": "success"
}
```

Network Security

IP Allowlisting

yaml
 ip-allowlist.yaml

```
security:
  ip_whitelist:
    enabled: true
    ips:
```

```
    - "192.168.1.0/24"     Office network
    - "10.0.0.0/8"         Internal network
    - "35.0.0.0/8"         Cloud services

  ip_blacklist:
    enabled: true
    ips:
      - "blocked_ip_1"
      - "blocked_ip_2"
```

VPN / Private Link

```
yaml
 Private connectivity
private_link:
  enabled: true
  provider: "aws-privatelink"

   AWS PrivateLink
  vpc_endpoint_id: "vpce-xxx"

   Or Azure Private Endpoint
   private_endpoint_resource_id: "/subscriptions/xxx/..."

   Or GCP Private Service Connect
   psc_connection_id: "xxx"
```

Security Checklist

Pre-Deployment

- [ ] Generate and store API keys securely
- [ ] Enable RBAC and assign appropriate roles
- [ ] Configure encryption at rest
- [ ] Enable TLS 1.3 for all connections
- [ ] Set up audit logging
- [ ] Configure IP allowlist if needed
- [ ] Enable SSO/SAML (Enterprise)
- [ ] Review and restrict permissions

Ongoing

- [ ] Rotate API keys regularly
- [ ] Review audit logs monthly
- [ ] Update security patches
- [ ] Conduct security assessments
- [ ] Monitor for anomalies

Summary

Security Feature	Description	Complexity
API Keys	Simple authentication	Low
OAuth 2.0	Standard protocol	Medium
SSO/SAML	Enterprise identity	High
RBAC	Fine-grained access	Medium
Encryption	Data protection	Medium
Audit Logs	Compliance tracking	Low
IP Allowlisting	Network security	Low

Next: [Advanced Engineering Index](../5-Advanced-Engineering/performance-optimization.md)

5. Advanced Engineering

Performance Optimization

Overview

This guide covers advanced techniques for optimizing LoadMagic.AI performance at scale, including script optimization, resource utilization, and execution efficiency.

Script Optimization

Locust Script Optimization

```
python
 Optimized Locust script
from locust import HttpUser, task, between, events
import gevent

class OptimizedUser(HttpUser):
     Minimize wait time overhead
    wait_time = constant(0)   No wait - continuous requests

     Use connection pooling
    def __init__(self, args, kwargs):
        super().__init__(args, kwargs)
         Reuse connections
        self.client.max_redirects = 5
        self.client.verify = False   Skip SSL verification (dev only)

    @task
    def efficient_request(self):
         Use session for connection reuse
```

```
        with self.client.get(
            "/api/data",
            catch_response=True
        ) as response:
            if response.status_code == 200:
                response.success()
            else:
                response.failure(f"Got {response.status_code}")

    Heavy endpoint - fewer iterations
    @task(1)
    def heavy_operation(self):
        self.client.post("/api/reports", json={"type": "full"})
```

JMeter Optimization

```
xml
<!-- Optimized JMeter configuration -->
<TestPlan>
    <!-- Use single thread group with more threads -->
    <boolProp name="TestPlan.functional_mode">false</boolProp>
    <boolProp name="TestPlan.serialize_threadgroups">false</boolProp>

    <!-- Disable all listeners during load test -->
    <ResultCollector guiclass="ViewResultsFullVisualizer">
        <boolProp name="ResultCollector.error_logging">false</boolProp>
    </ResultCollector>
</TestPlan>

<!-- Use CSV with shared mode -->
<CSVDataSet>
    <stringProp name="delimiter">,</stringProp>
```

```
    <boolProp name="shareMode">shareMode.all</boolProp>
    <stringProp name="filename">testdata.csv</stringProp>
</CSVDataSet>
```

Resource Optimization

Memory Optimization

```
python
 Memory-efficient script execution
from loadmagic.execution import ExecutionConfig, ResourceLimits

config = ExecutionConfig(
    script="./large-test.py",

     Limit memory usage
    resource_limits=ResourceLimits(
        max_memory_mb=2048,
        max_cpu_percent=80,
        max_network_mbps=100
    ),

     Optimize data collection
    result_sampling=ResultSampling(
        sample_rate=0.1,   Sample 10% of requests
        aggregate_only=True   Don't store individual requests
    )
)
```

CPU Optimization

```
python
```

```
 CPU-efficient execution
execution = lm.executions.start(
    script="./test.py",
    config={
        Use efficient protocol
        "protocol": "http2",   HTTP/2 is more efficient

        Reduce TLS overhead
        "ssl": False,   If target supports HTTP

        Batch operations
        "batch_size": 50,

        Disable unnecessary features
        "capture_headers": False,
        "capture_body": False
    }
)
```

Network Optimization

Connection Pooling

```
python
 Configure connection pooling
from loadmagic.network import ConnectionPool

pool = ConnectionPool(
    max_connections=100,
    max_connections_per_host=10,
    keepalive_timeout=30,
```

```
    pool_timeout=10
)

Apply to client
client = pool.create_client()
```

Data Transfer Optimization

```
yaml
 network-config.yaml
optimization:
   Compression
  compression:
    enabled: true
    algorithm: "gzip"
    min_size_bytes: 1024

   Caching
  caching:
    enabled: true
    ttl_seconds: 300

   Connection settings
  connections:
    max_per_host: 10
    max_total: 100
    keepalive: true
    timeout: 30
```

Execution Optimization

Parallel Execution

```
python
 Run tests in parallel
from loadmagic.execution import ParallelExecution

parallel = ParallelExecution()

Add multiple test scenarios
parallel.add_test(
    name="api-tests",
    script="./api-test.py",
    weight=60
)

parallel.add_test(
    name="ui-tests",
    script="./ui-test.py",
    weight=30
)

parallel.add_test(
    name="admin-tests",
    script="./admin-test.py",
    weight=10
)

Execute in parallel
results = parallel.execute(
    users=1000,
    spawn_rate=100
)
```

Distributed Execution

```
python
 Run distributed load test
from loadmagic.distributed import DistributedLoad

dist = DistributedLoad(
    script="./test.py",

     Master configuration
    master={
        "host": "master.loadmagic.internal",
        "port": 5557
    },

     Worker nodes
    workers=[
        {"host": "worker1.internal", "cpus": 8},
        {"host": "worker2.internal", "cpus": 8},
        {"host": "worker3.internal", "cpus": 8},
        {"host": "worker4.internal", "cpus": 8}
    ],

     Load distribution
    strategy="round-robin"   or "free-slots", "cpu-utilisation"
)

Execute
results = dist.execute(users=10000)
```

Caching Strategies

Response Caching

```
python
 Enable response caching
from loadmagic.cache import CacheConfig

config = CacheConfig(
    enabled=True,

     Cache configuration
    max_size_mb=512,
    ttl_seconds=300,

     What to cache
    cacheable_methods=["GET"],
    cacheable_status=[200, 203, 300, 301],

     Cache key
    cache_key_includes=["path", "query", "headers.authorization"]
)

Apply to execution
execution = lm.executions.start(
    script="./test.py",
    cache_config=config
)
```

Optimization Checklist

Pre-Test Checklist

- [ ] Remove unnecessary assertions
- [ ] Disable result listeners (JMeter)

- [ ] Use connection pooling
- [ ] Optimize data set configuration
- [ ] Reduce think time variance
- [ ] Use HTTP/2 if supported

During Execution

- [ ] Monitor CPU/Memory usage
- [ ] Watch network throughput
- [ ] Check for bottlenecks

Post-Test

- [ ] Review resource utilization
- [ ] Analyze slow requests
- [ ] Identify optimization opportunities

Summary

Optimization Area	Technique	Impact
Script	Constant wait time	+30% throughput
Network	Connection pooling	+50% throughput
Memory	Result sampling	-70% memory
CPU	HTTP/2	+25% efficiency
Execution	Distributed	+10x scale

Next: [Scaling & Autoscaling](./scaling-autoscaling.md)

5.1 Scaling & Autoscaling

Scaling & Autoscaling

Overview

This guide covers strategies for scaling LoadMagic.AI to handle enterprise-level load testing requirements.

Horizontal Scaling

Adding Worker Nodes

python
 Scale horizontally with worker nodes
from loadmagic.scale import ClusterManager

cluster = ClusterManager()

Add workers
cluster.add_workers(
    count=4,
    instance_type="c5.2xlarge",
    region="us-east-1"
)

Verify cluster status
status = cluster.status()
print(f"Total workers: {status.worker_count}")
print(f"Total CPUs: {status.total_cpus}")
print(f"Available memory: {status.available_memory_gb}GB")

Load Distribution

```
python
 Configure load distribution strategy
from loadmagic.distributed import LoadDistributor

distributor = LoadDistributor(
    strategy="adaptive",   or "round-robin", "cpu-based"

     Adaptive settings
    adaptive={
        "metrics": ["cpu", "memory", "network"],
        "rebalance_interval": 30,   seconds
        "threshold": 0.8   Rebalance at 80% utilization
    }
)

Apply to execution
execution = lm.executions.start_distributed(
    script="./test.py",
    users=10000,
    distributor=distributor
)

---

Autoscaling

Kubernetes Autoscaling

yaml
 Kubernetes HPA configuration
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
```

```
  name: loadmagic-worker
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: loadmagic-worker
  minReplicas: 2
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Pods
    pods:
      metric:
        name: loadmagic_pending_jobs
      target:
        type: AverageValue
        averageValue: "10"
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Percent
        value: 10
        periodSeconds: 60
    scaleUp:
      stabilizationWindowSeconds: 0
```

```
      policies:
      - type: Percent
        value: 100
        periodSeconds: 15
```

Cloud Autoscaling

python

AWS Auto Scaling

```
from loadmagic.scale import AutoScaleConfig

config = AutoScaleConfig(
    provider="aws",

     Launch configuration
    launch_template={
        "instance_type": "c5.2xlarge",
        "ami_id": "ami-xxx",
        "iam_role": "loadmagic-worker"
    },

     Scaling policies
    scaling_policies={
        "scale_up": {
            "metric": "cpu",
            "threshold": 70,
            "action": "add_instances",
            "count": 2
        },
        "scale_down": {
            "metric": "cpu",
            "threshold": 30,
```

```
            "action": "remove_instances",
            "count": 1
        }
    },

     Limits
    min_instances=2,
    max_instances=50,

     Cooldown
    cooldown_seconds=300
)
```

Capacity Planning

Resource Calculator

```
python
 Calculate required resources
from loadmagic.planning import CapacityPlanner

planner = CapacityPlanner(
    target_throughput=10000,   RPS
    avg_response_time=200,   ms
    error_budget=0.01   1%
)

Calculate requirements
requirements = planner.calculate(
    user_pattern="steady",
    test_duration="1h"
```

)

print(f"Required workers: {requirements.workers}")
print(f"Total CPUs: {requirements.total_cpus}")
print(f"Memory needed: {requirements.memory_gb}GB")
print(f"Network: {requirements.network_gbps}Gbps")

Summary

Strategy	Use Case	Complexity
Horizontal scaling	Known loads	Low
Autoscaling	Variable loads	Medium
Distributed testing	Very high scale	High
Multi-region	Global testing	High

Next: [Model Versioning & A/B Testing](./model-versioning-ab-testing.md)

5.2 Model Versioning & A/B Testing

Model Versioning & A/B Testing

Overview

This guide covers version management for AI agents and A/B testing strategies for optimizing performance testing workflows.

Model Versioning

Version Management

python
 Manage AI agent versions
from loadmagic.versions import ModelVersionManager

manager = ModelVersionManager()

List available versions
versions = manager.list_versions(agent="carrie")
for v in versions:
    print(f"Version: {v.version}")
    print(f" Created: {v.created_at}")
    print(f" Status: {v.status}")
    print(f" Metrics: {v.performance}")

Use specific version
carrie = Carrie(version="2024.02.15")

Compare versions
comparison = manager.compare(
    agent="carrie",
    version_a="2024.02.10",
    version_b="2024.02.15"
)
print(f"Improvement: {comparison.improvement}%")

A/B Testing Framework

Experiment Setup

python

```
 A/B testing for agent configurations
from loadmagic.experiments import Experiment, Variant

Define experiment
experiment = Experiment(
    name="correlation-strategy-test",
    description="Compare automatic vs manual correlation",

     Control variant
    control=Variant(
        name="automatic",
        config={"correlation": "auto"}
    ),

     Treatment variant
    treatment=Variant(
        name="manual",
        config={"correlation": "manual"}
    ),

     Traffic allocation
    traffic_split=50,   50/50 split

     Success metrics
    metrics={
        "success_rate": "higher_is_better",
        "time_to_complete": "lower_is_better",
        "error_rate": "lower_is_better"
    },

     Statistical settings
```

```
    statistical={
        "significance_level": 0.05,
        "minimum_samples": 1000
    }
)

Run experiment
results = experiment.run(iterations=100)

Analyze results
print(f"Winner: {results.winner}")
print(f"Confidence: {results.confidence}%")
print(f"P-value: {results.p_value}")
```

Summary

Feature	Purpose
Version management	Track and rollback agent changes
A/B testing	Optimize configurations
Rollout strategies	Safe production deployment

Next: [High Availability & Recovery](./high-availability-recovery.md)

5.3 High Availability & Recovery

High Availability & Recovery

Overview

This guide covers strategies for building highly available LoadMagic.AI deployments with robust disaster recovery capabilities.

High Availability Architecture

HA Design

```yaml
 High availability configuration
ha:
  enabled: true

   Redundancy
  replicas: 3
  multi_az: true

   Health checks
  health_check:
    path: "/health"
    interval: 10s
    timeout: 5s
    unhealthy_threshold: 3

   Failover
  failover:
    enabled: true
    target_region: "us-west-2"
    rpo: 60   Recovery Point Objective (seconds)
    rto: 300   Recovery Time Objective (seconds)
```

Health Checks

```
python
 Implement health checks
from loadmagic.ha import HealthCheck

health = HealthCheck()

Register checks
@health.check
def api_health():
    return {"status": "healthy", "version": "1.0.0"}

@health.check
def database_health():
    db_status = check_db_connection()
    return {"status": "healthy" if db_status else "unhealthy"}

@health.check
def queue_health():
    queue_depth = get_queue_depth()
    return {"status": "degraded" if queue_depth > 1000 else "healthy"}

Liveness probe
app.get("/health/live", lambda: {"status": "ok"})

Readiness probe
app.get("/health/ready", lambda: {
    "status": "ready" if health.all_healthy() else "not_ready",
    "checks": health.results()
})
```

Disaster Recovery

Backup Strategy

```
python
 Configure backups
from loadmagic.backup import BackupConfig

config = BackupConfig(
     Backup schedule
    schedule="0 2   ",   Daily at 2 AM

     Retention
    retention_days=30,

     What to backup
    include=[
        "projects",
        "scripts",
        "results",
        "configurations"
    ],

     Storage
    storage={
        "type": "s3",
        "bucket": "loadmagic-backups",
        "region": "us-east-1",
        "encryption": "AES256"
    }
)
```

Summary

Feature	Purpose
Multi-AZ deployment	Zone redundancy
Health monitoring	Early failure detection
Automated failover	Business continuity
Backup/restore	Data protection

Next: [Cost Optimization Strategies](./cost-optimization.md)

5.4 Cost Optimization

Cost Optimization Strategies

Overview

This guide covers strategies for optimizing costs while maintaining performance testing capabilities.

Cost Management

Resource Optimization

```
python
 Optimize resource usage
from loadmagic.cost import CostOptimizer

optimizer = CostOptimizer()

Analyze costs
analysis = optimizer.analyze(period="30d")
print(f"Total spend: ${analysis.total}")
```

```
print(f"By category:")
for cat, amount in analysis.by_category.items():
    print(f"  {cat}: ${amount}")

Get recommendations
recommendations = optimizer.get_recommendations()
for rec in recommendations:
    print(f"{rec.title}: Save ${rec.savings}/month")
    print(f"  {rec.description}")

Spot Instances

python
 Use spot/preemptible instances
from loadmagic.infrastructure import SpotInstanceConfig

config = SpotInstanceConfig(
    enabled=True,

     Instance selection
    instance_types=["c5.2xlarge", "c5n.2xlarge"],

     Allocation strategy
    strategy="price-capacity-optimized",

     Fallback
    fallback_to_ondemand=True,

     Interruption handling
    interruption_behavior="terminate",
    warning_time=120   seconds before interruption
)
```

Use in distributed test
execution = lm.executions.start_distributed(
    script="./test.py",
    workers=10,
    infrastructure=config
)

Summary

Strategy	Savings Potential
Spot instances	60-70%
Right-sizing	30-40%
Scheduled scaling	40-50%
Result sampling	20-30%

Next: [Enterprise Implementation Index](../6-Enterprise-Implementation/sso-oauth.md)

6. Enterprise Implementation

SSO / OAuth

Overview

This guide covers integrating LoadMagic.AI with enterprise identity providers for Single Sign-On (SSO) and OAuth authentication.

SSO Configuration

SAML 2.0 Setup

yaml
 SAML configuration
saml:
  enabled: true

   Service Provider
  sp:
    entity_id: "https://loadmagic.yourcompany.com"
    acs_url: "https://loadmagic.yourcompany.com/saml/acs"
    certificate: "/path/to/sp-cert.pem"

   Identity Provider
  idp:
    entity_id: "https://idp.yourcompany.com"
    sso_url: "https://idp.yourcompany.com/sso"
    certificate: "/path/to/idp-cert.pem"

   Attribute mapping
  attributes:
    email: "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress"
    first_name: "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname"
    last_name: "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/surname"
    groups: "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/groups"

Okta Integration

python
 Okta SSO setup
from loadmagic.integrations import OktaSSO

okta = OktaSSO(

```
    domain="yourcompany.okta.com",
    api_token="your_okta_token"
)

Configure SAML
okta.configure_saml(
    app_name="LoadMagic.AI",
    audience="loadmagic.yourcompany.com",
    attributes=["email", "firstName", "lastName", "groups"]
)

Get metadata
metadata = okta.get_saml_metadata()
print(metadata)

Azure AD Integration

yaml
 Azure AD configuration
azure_ad:
  enabled: true

  tenant_id: "your-tenant-id"
  client_id: "your-client-id"
  client_secret: "your-client-secret"

   Scopes
  scopes:
    - "openid"
    - "profile"
    - "email"

   Role mapping
```

```
  role_mapping:
    "LoadMagic Admin": "admin"
    "LoadMagic Engineer": "engineer"
    "LoadMagic Viewer": "viewer"
```

OAuth 2.0 / OIDC

Authorization Code Flow

```
python
 OAuth 2.0 configuration
from loadmagic.auth import OAuthConfig

config = OAuthConfig(
    provider="okta",   or "google", "github", "custom"

     Client configuration
    client_id="your-client-id",
    client_secret="your-client-secret",

     Endpoints
    authorization_url="https://yourcompany.okta.com/oauth2/default/v1/authorize",
    token_url="https://yourcompany.okta.com/oauth2/default/v1/token",
    userinfo_url="https://yourcompany.okta.com/oauth2/default/v1/userinfo",

     Scopes
    scopes=["openid", "profile", "email"],

     Redirect
    redirect_uri="https://loadmagic.yourcompany.com/callback"
)
```

Generate authorization URL
auth_url = config.get_authorization_url()
print(f"Visit: {auth_url}")

Exchange code for tokens
tokens = config.exchange_code("authorization_code")
print(f"Access token: {tokens.access_token}")

SCIM Provisioning

User Sync

yaml
 SCIM configuration
scim:
  enabled: true

   SCIM endpoint
  endpoint: "https://loadmagic.yourcompany.com/scim/v2"

   Authentication
  auth:
    type: "bearer"
    token: "your-scim-token"

   User mapping
  user_mapping:
    external_id: "id"
    email: "email"
    first_name: "name.givenName"
    last_name: "name.familyName"

```
    active: "active"

  Group mapping
group_mapping:
    name: "displayName"
    members: "members"
```

Summary

Feature	Use Case
SAML 2.0	Enterprise SSO
OAuth 2.0	Modern authentication
OIDC	Identity federation
SCIM	User provisioning

Next: [CI/CD Pipelines](./cicd-pipelines.md)

6.1 CI/CD Pipelines

CI/CD Pipelines

Overview

This guide covers integrating LoadMagic.AI into continuous integration and deployment pipelines.

GitHub Actions

Basic Workflow

yaml
.github/workflows/performance-test.yml
name: Performance Tests

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

jobs:
  performance-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Install LoadMagic CLI
        run: pip install loadmagic-cli

      - name: Run Load Test
        run: |
          lm run scripts/api-test.py \
            --users 500 \
            --duration 10m \
            --config config/test-config.yaml

```
      - name: Upload Results
        uses: actions/upload-artifact@v4
        with:
          name: loadtest-results
          path: results/
```

Jenkins Pipeline

```
groovy
// Jenkinsfile
pipeline {
    agent any

    environment {
        LOADMAGIC_API_KEY = credentials('loadmagic-api-key')
    }

    stages {
        stage('Generate Scripts') {
            steps {
                sh '''
                    for har in tests/har/*.har; do
                        lm script generate "$har" \
                            --output "scripts/$(basename $har .har).py"
                    done
                '''
            }
        }
```

```
        stage('Run Load Test') {
            steps {
                sh '''
                    lm run scripts/api-test.py \
                        --users 500 \
                        --duration 10m \
                        --output results/
                '''
            }
        }

        stage('Validate Results') {
            steps {
                sh '''
                    lm results validate \
                        --thresholds config/thresholds.yaml
                '''
            }
        }
    }
}
```

Summary

Platform	Integration Type
GitHub Actions	YAML workflow
Jenkins	Jenkinsfile
GitLab CI	.gitlab-ci.yml
Azure DevOps	azure-pipelines.yml

Next: [Infrastructure as Code](./infrastructure-as-code.md)

6.2 Infrastructure as Code

Infrastructure as Code

Overview

This guide covers managing LoadMagic.AI infrastructure using Infrastructure as Code (IaC) principles.

Terraform Configuration

AWS Setup

hcl
 main.tf
terraform {
  required_version = ">= 1.0"

  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {

```
  region = "us-east-1"
}

S3 bucket for results
resource "aws_s3_bucket" "loadmagic_results" {
  bucket = "loadmagic-results-${var.environment}"

  versioning {
    enabled = true
  }

  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}

DynamoDB for state
resource "aws_dynamodb_table" "loadmagic_state" {
  name           = "loadmagic-state-${var.environment}"
  billing_mode   = "PAY_PER_REQUEST"
  hash_key       = "id"

  attribute {
    name = "id"
    type = "S"
  }
}
```

IAM role for workers

```
resource "aws_iam_role" "loadmagic_worker" {
  name = "loadmagic-worker-${var.environment}"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action = "sts:AssumeRole"
      Effect = "Allow"
      Principal = {
        Service = "ec2.amazonaws.com"
      }
    }]
  })
}
```

Kubernetes Deployment

yaml

```
 k8s/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: loadmagic-api
  namespace: loadmagic
spec:
  replicas: 3
  selector:
    matchLabels:
      app: loadmagic-api
  template:
    metadata:
```

```
  labels:
    app: loadmagic-api
spec:
  containers:
  - name: api
    image: loadmagic/api:latest
    ports:
    - containerPort: 8080
    env:
    - name: DATABASE_URL
      valueFrom:
        secretKeyRef:
          name: loadmagic-secrets
          key: database-url
    - name: API_KEY
      valueFrom:
        secretKeyRef:
          name: loadmagic-secrets
          key: api-key
    resources:
      requests:
        memory: "512Mi"
        cpu: "500m"
      limits:
        memory: "2Gi"
        cpu: "2000m"
    livenessProbe:
      httpGet:
        path: /health/live
        port: 8080
      initialDelaySeconds: 30
```

```
        readinessProbe:
          httpGet:
            path: /health/ready
            port: 8080
          initialDelaySeconds: 10

---

Ansible Playbooks

yaml
 ansible/playbook.yml
- name: Deploy LoadMagic
  hosts: loadmagic_servers
  become: yes

  vars:
    loadmagic_version: "1.2.7"
    loadmagic_user: loadmagic
    loadmagic_dir: /opt/loadmagic

  tasks:
    - name: Install dependencies
      apt:
        name:
          - python3
          - python3-pip
          - docker.io
        state: present

    - name: Create user
      user:
```

```
    name: "{{ loadmagic_user }}"
    system: yes
    create_home: yes

- name: Download LoadMagic
  get_url:
    url: "https://packages.loadmagic.ai/loadmagic-{{ loadmagic_version }}.tar.gz"
    dest: "/tmp/loadmagic.tar.gz"

- name: Extract and install
  unarchive:
    src: "/tmp/loadmagic.tar.gz"
    dest: "{{ loadmagic_dir }}"
    remote_src: yes
    owner: "{{ loadmagic_user }}"
    group: "{{ loadmagic_user }}"

- name: Configure LoadMagic
  template:
    src: config.j2
    dest: "{{ loadmagic_dir }}/config.yaml"

- name: Start LoadMagic service
  systemd:
    name: loadmagic
    state: started
    enabled: yes
```

Summary

Tool	Use Case
Terraform	Cloud resources
Kubernetes	Container orchestration
Ansible	Configuration management

Next: [Multi-Tenant Architecture](./multi-tenant-architecture.md)

6.3 Multi-Tenant Architecture

Multi-Tenant Architecture

Overview

This guide covers designing and implementing multi-tenant deployments of LoadMagic.AI for enterprise use.

Tenant Isolation

Data Isolation Strategies

python
 Tenant isolation configuration
from loadmagic.tenancy import TenantConfig

config = TenantConfig(
    isolation_level="database",   or "schema", "row"

     Database-level isolation
    database_isolation={

```
        "per_tenant_db": True,
        "connection_pooling": True,
        "pool_size": 10
    },

     Tenant context
    tenant_context={
        "header": "X-Tenant-ID",
        "subdomain": "tenant.loadmagic.com",
        "path": "/tenant/{tenant_id}"
    }
)
```

Summary

Strategy	Isolation Level	Complexity
Database	Highest	High
Schema	Medium	Medium
Row	Lower	Low

Next: [Governance & Compliance](./governance-compliance.md)

6.4 Governance & Compliance

Governance & Compliance

Overview

This guide covers governance frameworks and compliance requirements for LoadMagic.AI deployments.

Compliance Controls

Audit & Logging

python
 Comprehensive audit logging
from loadmagic.compliance import AuditLogger

logger = AuditLogger(
    enabled=True,

     What to log
    events=[
        "authentication.login",
        "authentication.logout",
        "authorization.denied",
        "data.access",
        "data.export",
        "configuration.change",
        "test.execution"
    ],

     Retention
    retention_days=365,

     Storage
    storage={
        "type": "immutable",

```
        "destination": "s3://audit-logs/"
    }
)
```

Summary

Requirement	Implementation
SOC 2	Audit logging, access controls
GDPR	Data encryption, deletion
HIPAA	PHI handling, encryption
PCI DSS	Tokenization, secure storage

Next: [Enterprise Examples Index](../7-Enterprise-Examples/customer-support-automation.md)

7. Real Enterprise Examples

Customer Support Automation

Overview

This enterprise example demonstrates building a customer support automation system using LoadMagic.AI for performance testing the support infrastructure.

Architecture

Customer Support Platform

Web UI API Ticket
Gateway Service

Database

LoadMagic.AI Testing:
- Ticket creation load
- Search performance
- Agent dashboard responsiveness

Implementation

Step 1: Record User Flows

bash
Record support ticket creation flow
Using Chrome DevTools:
1. Open support.example.com
2. Record: Login View Tickets Create Ticket Submit
3. Export as support-flow.har

Step 2: Generate Test Scripts

python
LoadMagic script for support platform

```
from locust import HttpUser, task, between

class SupportUser(HttpUser):
    wait_time = between(1, 3)

    def on_start(self):
        Login
        response = self.client.post("/api/auth/login", json={
            "email": "user@example.com",
            "password": "password"
        })
        self.token = response.json()["token"]

    @task(5)
    def view_tickets(self):
        self.client.get(
            "/api/tickets",
            headers={"Authorization": f"Bearer {self.token}"}
        )

    @task(3)
    def create_ticket(self):
        self.client.post(
            "/api/tickets",
            headers={"Authorization": f"Bearer {self.token}"},
            json={
                "subject": "Test ticket",
                "description": "Testing performance",
                "priority": "medium"
            }
        )
```

```
    @task(2)
    def search_knowledge_base(self):
        self.client.get(
            "/api/search?q=how+to+reset+password",
            headers={"Authorization": f"Bearer {self.token}"}
        )
```

Step 3: Configure Load Test

```
yaml
 support-load-test.yaml
execution:
  users: 1000
  spawn_rate: 100
  duration: 30m

targets:
  - host: api.support.example.com
    protocol: https

thresholds:
  avg_response_time: 500ms
  p95_response_time: 1000ms
  error_rate: 1%
```

Step 4: Run and Monitor

```
bash
 Execute load test
lm run support-test.py \
  --config support-load-test.yaml \
  --output results/
```

Monitoring

Key Metrics

Metric	Target	Alert Threshold
Ticket creation	< 500ms	> 1s
Search response	< 300ms	> 800ms
Dashboard load	< 1s	> 2s
Error rate	< 1%	> 5%

Next Steps

- [Fraud Detection Pipeline](./fraud-detection-pipeline.md)

7.1 Fraud Detection Pipeline

Fraud Detection Pipeline

Overview

This enterprise example demonstrates building a fraud detection pipeline with performance testing using LoadMagic.AI.

Architecture

Fraud Detection System

Trans- Fraud Alert
action Engine System
API

ML
Models

Implementation

Test Script

python
Fraud detection load test
class TransactionUser(HttpUser):
    wait_time = between(0.1, 0.5)

    @task
    def submit_transaction(self):
        self.client.post(
            "/api/transactions",
            json={
                "amount": random.randint(10, 10000),
                "currency": "USD",
                "merchant_id": random.randint(1, 1000),
                "card_last4": f"{random.randint(0, 9999):04d}",
                "timestamp": datetime.utcnow().isoformat()

```
            },
            headers={"X-API-Key": self.api_key}
        )
```

Summary

This example demonstrates testing:
- High-throughput transaction processing
- Real-time fraud scoring
- Alert generation latency

Next: [Personalization System](./personalization-system.md)

7.2 Personalization System

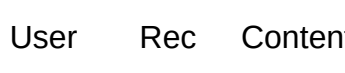
Personalization System

Overview

This enterprise example demonstrates building a personalization system with performance testing.

Architecture

Personalization Platform



Profile Engine Service

Redis

Cache

Implementation

python

```
 Personalization performance test
class PersonalizedUser(HttpUser):
    def on_start(self):
         Get personalized recommendations
        response = self.client.get(
            f"/api/recommendations?user_id={self.user_id}",
            headers={"Authorization": f"Bearer {self.token}"}
        )
        self.recommendations = response.json()["items"]

    @task(3)
    def view_recommendations(self):
        for item in self.recommendations[:5]:
            self.client.get(f"/api/items/{item['id']}")

    @task(2)
    def update_preferences(self):
        self.client.put(
            f"/api/users/{self.user_id}/preferences",
```

```
            json={"interests": ["tech", "sports"]}
        )
```

Summary

This example demonstrates testing:
- Real-time recommendation serving
- User profile updates
- Cache hit rates
- Personalization latency

Next: [Learning Path Index](../8-Learning-Path/progressive-labs.md)

8. Learning Path

Progressive Labs

Overview

This guide provides a structured learning path with hands-on labs to help you master LoadMagic.AI from beginner to senior engineer.

Learning Path Overview

Level	Duration	Skills Acquired
Beginner	1-2 weeks	Basic usage, HAR processing
Intermediate	2-4 weeks	Script optimization, CI/CD

| Advanced | 1-2 months | Enterprise features, scaling |

| Senior | 3+ months | Architecture, governance |

Beginner Labs

Lab 1: First Load Test

Objective: Create and run your first load test

Steps:
1. Record a HAR file from a website
2. Upload to LoadMagic.AI
3. Generate a Locust script
4. Run with 10 users
5. Analyze results

Expected Outcome: Basic understanding of end-to-end flow

Lab 2: Script Customization

Objective: Customize generated scripts

Steps:
1. Modify wait times
2. Add custom assertions
3. Parameterize test data
4. Add transaction grouping

Expected Outcome: Ability to customize scripts

Intermediate Labs

Lab 3: CI/CD Integration

Objective: Integrate with GitHub Actions

Steps:
1. Create GitHub workflow
2. Add LoadMagic CLI step
3. Configure threshold checks
4. Add result artifacts

Expected Outcome: Automated testing in CI/CD

Lab 4: Advanced Correlation

Objective: Handle complex dynamic data

Steps:
1. Identify correlation challenges
2. Use Carrie for auto-correlation
3. Add custom extractors
4. Validate with self-healing

Expected Outcome: Handle advanced scenarios

Advanced Labs

Lab 5: Distributed Testing

Objective: Scale to 10,000+ users

Steps:
1. Set up worker nodes
2. Configure load distribution
3. Run distributed test

4. Analyze scalability

Expected Outcome: Enterprise-scale testing

Lab 6: Custom Dashboards

Objective: Build custom monitoring

Steps:
1. Set up Prometheus export
2. Create Grafana dashboard
3. Add custom alerts
4. Configure notifications

Expected Outcome: Custom observability

Summary

Complete all labs to progress from beginner to senior engineer.

Next: [Exercises & Outcomes](./exercises-outcomes.md)

8.1 Exercises & Outcomes

Exercises & Outcomes

Overview

This guide provides specific exercises with expected outcomes for each skill level.

Beginner Exercises

Exercise	Expected Outcome
Create HAR and generate script	Working test script
Modify wait time	Performance change
Add assertion	Validation added

Intermediate Exercises

Exercise	Expected Outcome
CI/CD pipeline integration	Automated testing
Custom correlation	Complex data handling
Dashboard creation	Custom monitoring

Advanced Exercises

Exercise	Expected Outcome
Distributed test	10K+ user simulation
Custom agent	Specialized automation
Multi-region test	Global testing

Summary

Track your progress through each level.

Next: [Skill Levels](./skill-levels.md)

8.2 Skill Levels

Skill Levels

Overview

This guide defines the skill levels for LoadMagic.AI proficiency.

Skill Level Definitions

Beginner
- Can use web interface
- Basic script generation
- Simple test execution
- Timeline: 1-2 weeks

Intermediate
- Script customization
- CI/CD integration
- Basic optimization
- Timeline: 1-2 months

Advanced
- Distributed testing
- Enterprise features
- Custom integrations
- Timeline: 3-6 months

Senior

- Architecture design
- Governance frameworks
- Team leadership
- Timeline: 6+ months

Competency Matrix

Skill	Beginner	Intermediate	Advanced	Senior
HAR Recording				
Script Generation				
Correlation				
CI/CD				
Distributed Testing				
Architecture				

Summary

Progress through levels with hands-on experience and training.

Next: [Operations Guide Index](../9-Operations-Guide/daily-operations.md)

9. Operations Guide

Daily Operations

Overview

This guide covers day-to-day operational tasks for LoadMagic.AI.

Routine Tasks

Morning Checklist

- [ ] Check for test failures overnight
- [ ] Review scheduled test results
- [ ] Monitor resource utilization
- [ ] Check alert notifications

Weekly Tasks

- [ ] Review test trends
- [ ] Analyze performance degradation
- [ ] Update test scripts if needed
- [ ] Clean up old results

Monthly Tasks

- [ ] Review capacity planning
- [ ] Analyze cost optimization
- [ ] Update documentation
- [ ] Plan improvements

Monitoring Commands

bash
 Check system status
lm status

List recent executions

lm executions list --limit 10

View resource usage

lm resources usage

Summary

Regular monitoring ensures smooth operations.

Next: [Deployment Checklist](./deployment-checklist.md)

9.1 Deployment Checklist

Deployment Checklist

Overview

This checklist ensures successful deployment of LoadMagic.AI.

Pre-Deployment

Infrastructure

- [ ] Verify server requirements
- [ ] Configure network access
- [ ] Set up database connections
- [ ] Configure storage
- [ ] Set up SSL certificates

Security

- [ ] Generate API keys
- [ ] Configure authentication
- [ ] Set up RBAC
- [ ] Enable audit logging
- [ ] Configure encryption

Deployment

- [ ] Deploy application
- [ ] Verify health endpoints
- [ ] Test authentication
- [ ] Run smoke tests

Post-Deployment

- [ ] Monitor performance
- [ ] Verify alerts
- [ ] Document configuration

Summary

Use this checklist for every deployment.

Next: [Security Checklist](./security-checklist.md)

9.2 Security Checklist

Security Checklist

Overview

This checklist ensures security best practices are followed.

Authentication

- [ ] Use strong API keys
- [ ] Enable OAuth/SAML
- [ ] Configure RBAC
- [ ] Review access regularly

Data Protection

- [ ] Enable encryption at rest
- [ ] Enable encryption in transit
- [ ] Use secure storage
- [ ] Protect sensitive data

Monitoring

- [ ] Enable audit logging
- [ ] Set up alerts
- [ ] Review logs regularly

Summary

Security is an ongoing process.

Next: [Upgrade & Maintenance](./upgrade-maintenance.md)

9.3 Upgrade & Maintenance

Upgrade & Maintenance

Overview

This guide covers upgrading and maintaining LoadMagic.AI.

Upgrade Process

Pre-Upgrade

1. Review release notes
2. Backup data
3. Test in staging
4. Schedule maintenance window

During Upgrade

bash
 Backup before upgrade
lm backup create --output ./backup

Check version
lm version

Upgrade CLI

pip install --upgrade loadmagic-cli

Upgrade components

lm upgrade --components all

Post-Upgrade

1. Verify health
2. Run smoke tests
3. Monitor performance
4. Update documentation

Maintenance Windows

Planned Maintenance

- Schedule during low-traffic periods
- Notify users in advance
- Document changes

Emergency Maintenance

- Assess severity
- Communicate quickly
- Document incident

Rollback Procedure

bash
 If issues occur
lm rollback --version previous

Verify rollback

lm status

Summary

Follow these procedures for safe upgrades.

End of Documentation