AutoPilot Handover Documentation

SYSTEM ARCHITECTURE & TECHNICAL OPERATION GUIDE

Document Type:	Engineering Handover Guide	**Status:**	Final / Production Ready
Target System:	Distributed Cloud Execution Engine	**Date:**	June 30, 2026

This document serves as the complete, definitive handover guide for the **AutoPilot** project. It comprehensively details the core system architecture, base technology stack, operational feature sets, operational workflows, local engineering development guidelines, enterprise secrets management, and a systematic engineering roadmap for scaling the ecosystem to seamlessly support millions of active users in a production environment.

1. System Design and Architecture

1.1 High-Level Architecture

AutoPilot is engineered as a highly distributed, pull-based cloud execution engine purposefully built to coordinate and execute automated browser testing processes at scale. The systemic layout strictly implements a decoupling of components orchestrated through a **Coordinator-Worker** pattern:

- **Coordinator (Master Backend):** Serving as the centralized operational core of the topology, this component governs overall application state. It manages enterprise-level user authentication, executes CRUD workflows for granular test configurations, orchestrates scheduling routines, and processes centralized system job queuing matrices.
- **Workers (Local Agents):** Operating as independent, decentralized external execution engines, these components continuously poll the master backend coordinator for newly queued operations. Agents run fully sandboxed headless browser test sessions (such as Selenium or Playwright environments), capture raw output artifacts (including localized page screenshots), route them directly up to centralized cloud storage layers, and provide structured execution state telemetry back to the coordinator block.
- **Frontend UI:** A responsive, high-performance React-based Single Page Application (SPA) providing visual control suites allowing engineering stakeholders to dynamically configure tests, scrutinize live execution runs, and trace runtime agent status.

1.2 Tech Stack

The implementation stack is explicitly configured across the following engineering domains:

Backend Infrastructure Java 21, Spring Boot 3.4.0, Maven Build Tooling	**Database & Migration** PostgreSQL 15, Flyway Automated Schema Migration Tooling
Frontend Topology React.js, Vite Build Pipeline, Static Delivery via Nginx	**Cloud Blob Storage** Amazon Simple Storage Service (Amazon S3)
Security & Auth Spring Security Framework with Stateless JSON Web Tokens (JWT)	**Containerization** Docker Engine, Multi-Container orchestration via Docker Compose

2. Features and Workflow

2.1 Core Features

- **Test Cases & Steps:** The atomic logical execution block of the testing lifecycle. Engineers define sequence matrices of contextual actions (e.g., clicks, user inputs, explicit page navigations). Steps natively support a custom `CALL_COMPONENT` flag, allowing nested architectural reusability of composite test cases across configurations.
- **Test Case Groups:** Structured logic containers enabling the bundling of independent Test Cases to be sequentially executed back-to-back within unified context bands.
- **Test Suites:** Representing the pinnacle of the scheduling configuration hierarchy. A suite links multiple Test Case Groups and allows declarative environmental targeting and direct task mapping to explicit Agent Groups.
- **Environments & Variables:** Comprehensive support for systemic global variable bindings and granular environment-specific variable overriding (e.g., swapping API route targets contextually between Staging and Production URLs).
- **Schedulers:** Flexible operational execution switches that enable ad-hoc "Run Now" processing or declarative scheduling blocks leveraging standard Outlook-style intervals (Once, Daily, Weekly, Monthly) as well as legacy cron expressions.
- **Agent Management:** Dynamic registration, health routing, and logical categorization of isolated pulling agents. Agents explicitly register target browser configurations, platform versions, and custom metadata.

2.2 Application Workflow

1. **Authoring:** An engineer structures Test Cases, maps them tightly inside relevant Groups, and registers the unified workflow to an over-arching Test Suite through the web UI frontend.

2. **Scheduling:** The scheduler maps execution configurations. The internal `CronExecutionEngine` component living on the backend dynamically evaluates complex multi-timezone configurations and operational trigger intervals.
3. **Queueing:** Upon scheduling trigger conditions being met, the master backend instantiates an immutable `Execution` database record and dynamically unwraps the parent Suite into highly granular atomic `Job` elements mapped in a `QUEUED` state.
4. **Polling & Execution:** Decentralized Local Agents continuously poll the core master api boundary using the `GET /api/agents/jobs/next` route block. If a queued item lines up perfectly with an agent's registered profile (browser engine parameters, specified Group ID metrics), the system locks down the job record and binds it explicitly to that agent.
5. **Artifacts:** The agent initializes the sandboxed headless browser to execute target actions. If visual screenshot triggers fail or complete, the agent automatically captures the frame assets and pushes them directly out to Amazon S3 via multi-part data streams.
6. **Completion:** Upon final step execution, the local worker assembles structural results metrics and transmits the payload directly via a single `POST /api/agents/executions/{id}/results` request back to the system core.

3. Secrets Management and Configuration

All sensitive operational parameters, cryptographic keys, and database passwords must follow structural separation rules and are externalized via environment variables.

3.1 Where Secrets are Stored

- **Local Development Environments:** Operational configurations are managed via a localized `.env` file structured directly within the project's root workspace directory (`/autopilot/.env`). This specific configuration asset is hard-locked within the global `.gitignore` rule-base to strictly block downstream code repository tracking.
- **Production Environments:** Secrets must be injected directly into runtime container contexts using an enterprise-grade cloud key vault ecosystem (such as AWS Secrets Manager, HashiCorp Vault, GitHub Encrypted Secrets, or native Kubernetes Secret manifests).

3.2 How to Modify Configurations

To dynamically adjust environment parameters like localized persistence credentials, security keys, or AWS connectivity setups within a local engineering framework, modify the root `.env` structure using these standardized keys:

ENVIRONMENT ENVIRONMENT CONFIGURATIONS (.ENV)

```
# Database Configuration
DB_NAME=autopilot_db
DB_USER=autopilot
```

```
DB_PASS=your_secure_password_here

# JWT Authentication
JWT_SECRET=your_very_long_base64_encoded_secret_key_here

# AWS S3 Configuration (For Screenshots)
AWS_REGION=us-east-1
AWS_ACCESS_KEY_ID=your_aws_access_key
AWS_SECRET_ACCESS_KEY=your_aws_secret_key
S3_BUCKET_NAME=your-s3-bucket-name
```

> **Configuration Injection Architecture:** The primary `docker-compose.yml` definition is pre-wired to extract these localized environment variables and map them cleanly into the Spring Boot backend container context (via `application-prod.yml` mappings) and the persistent PostgreSQL database instance layers at launch.

4. Local Setup and Execution

Follow this exact sequential setup layout to safely bootstrap and run the full technical workspace stack inside a new development environment:

1. **System Prerequisites:** Verify that local machines have a running, up-to-date instance of Docker Desktop and standard Git version control utilities configured.
2. **Clone the Target Repository:** Checkout the latest structural codebase iteration matching the production reference branch:

   ```
   git clone <repository-url>
   cd autopilot
   ```

3. **Instantiate Environment Variable Assets:** The docker container orchestration layout requires an explicit configuration manifest named exactly `.env`. Replicate the provided project skeleton template using platform-appropriate shell commands:

   PowerShell (Windows Environments):

   ```
   Copy-Item .env.example .env
   ```

   Bash / Mac / Linux Environments:

   ```
   cp .env.example .env
   ```

Open the newly instantiated `.env` configuration asset and accurately populate placeholders with valid development storage credentials and localized environment keys.

4. **Execute Codebase Assembly and Orchestration:** Run the container bootstrapping environment from the base `/autopilot` folder structure:

```
docker-compose up -d --build
```

> **Troubleshooting Memory Resource Starvation (Docker Crash/EOF Error):** If compiling the localized enterprise Java backend context concurrently with the Node-based React web application exhausts host resources, forcing a container termination or generating an unexpected `EOF` exception, bypass concurrent compilation steps by sequencing container builds as follows:
>
> ```
> docker-compose build master
> docker-compose build ui
> docker-compose up -d
> ```

4.3 System Access Mapping Boundaries

- **Frontend User Interface:** Accessible locally via http://localhost:80 (or standard port variations http://localhost).
- **Backend Gateway Core API:** Accessible locally via http://localhost:9090.
- **Persistence Database Core:** Network port `5432` is completely internal to the private virtualized Docker Compose software-defined network for security compliance. To connect an external database administration tool (such as DBeaver or DataGrip), explicitly map the database ports block to the host machine in the `docker-compose.yml` manifest.

5. Future Development (Pre-Go-Live Roadmap)

Prior to executing a definitive public deployment launch sequence, the incoming engineering unit must systematically resolve the following feature implementations and address underlying engineering debt items:

- **Agent Communication Security Core:** Refactor the baseline architecture to leverage persistent, secure WebSockets (`wss://` protocols) or standardized long-polling mechanisms to phase out high-frequency aggressive HTTP polling routines. This architectural shift will dramatically reduce server-side thread constraints. Additionally, build secure individual Agent API tokens for non-interactive worker registration and execution authorization.
- **Advanced Telemetry & Visual Dashboards:** Expand the React UI capabilities to embed structural tracking analytics, pass/fail trend vectors, execution duration scatterplots, and exportable data containers (such as native PDF execution certificates and raw CSV data matrix reports).

- **Native CI/CD Integration Pipelines:** Design and expose scalable webhook layers or construct marketplace action modules tailored directly for GitHub Actions, Jenkins Core, and GitLab CI frameworks to let pipelines programmatically fire off AutoPilot testing execution suites on code push events.
- **Multi-Tenant Concurrency Governance:** Build multi-tenant, enterprise-grade runtime rate limits along with logical concurrent test-job constraints globally to stop execution workloads from overwhelming shared computing nodes.
- **Automated Artifact Lifecycles:** Define and wire up native Amazon S3 Object Lifecycle Policies configured to systematically delete stale test screenshot blobs older than 30 retention days, keeping production cloud infrastructure spend stable.

6. Production Deployment (Scaling for Millions of Users)

To safely scale the AutoPilot platform to easily accommodate enterprise-level concurrent execution loads and millones of tracking sessions, the infrastructure layout must undergo a core shift away from single-host Docker Compose environments toward a resilient, highly available cloud-native microservices architecture.

6.1 Orchestration & Compute (AWS EKS)

Decommission the single-node runtime layer entirely. Migrate and package both the master Java backend service layer and the unified UI web presentation tier into lean, immutable, state-free microservice containers orchestrated cleanly through Kubernetes (via Amazon EKS). Implement Horizontal Pod Autoscaling (HPA) frameworks to programmatically spin compute resources up or down dynamically tracking real-time CPU consumption and RAM saturation.

6.2 Asynchronous Event Architecture (Kafka / RabbitMQ)

System Critical Constraint: The initial architectural layout depends entirely on transactional polling routines tracking the relational `jobs` table inside the core PostgreSQL database. Under high transactional production scaling, this design pattern triggers immediate write-lock collisions, thread pools exhaustion, and system-wide database degradation.

Target Architecture: Introduce a hardened event-streaming backbone (such as Apache Kafka or Amazon SQS). As schedulers fire, the master backend drops transactional jobs directly as immutable events onto highly partitioned messaging streams. Remote testing agents consume execution instructions directly from the distributed event streaming layer, completely eliminating state-tracking pressure from the relational storage core.

6.3 Storage Architecture Scaling (Amazon RDS PostgreSQL)

Migrate localized persistence architectures out to fully managed Amazon Aurora PostgreSQL database instances configured across multiple availability zones. Decouple transactional query behavior by creating dedicated database connection topologies:

- **Primary Write Node:** Exclusively processes incoming agent telemetry, execution state transitions, job state modifications, and core entity creation updates.
- **Distributed Read Replicas:** Dynamically handle high-volume user data queries, user dashboard interfaces, systemic telemetry visual structures, and read-only test configuration tracking.

6.4 High-Performance Caching (Amazon ElastiCache for Redis)

Integrate a lightning-fast distributed memory caching layer (Redis) positioned in front of the application tiers to serve as a specialized high-speed storage buffer for tracking live user session tokens, caching unchanging global configuration parameters, mapping runtime environmental variables, and validating tenant API rate-limiting thresholds at the gateway level.

6.5 UI Global Edge Processing (CDN Infrastructure)

The static React.js single-page web asset bundle must be moved off containerized application nodes entirely. Host all front-end compilation outputs directly inside an encrypted, highly secure Amazon S3 static website bucket fronted by Amazon CloudFront edge routing nodes. This pattern enforces rapid low-latency UI delivery around the world while removing static asset delivery processing overhead from application servers.

6.6 Enterprise Observability Suite

Deploy end-to-end telemetry frameworks (such as Datadog, New Relic, or a standardized Prometheus and Grafana stack). Engineers must implement distributed tracing layers driven by OpenTelemetry specifications to chart the complete lifecycle of individual execution threads: spanning from the immediate moment a user clicks "Run" on the client-side UI, processing across backend gateway routers, routing into the message broker, streaming down to decentralized local execution workers, and recording the eventual result loopback.