Project Scoping Submission - Otto

Team Members

- Sahil Chute
- Paschal Corrigan
- Tom Howes
- Malav Patel
- Shloka Shreyans Trivedi
- Ayushman Khandelwal

1. Introduction

Otto is a software engineering project management solution for agile teams that utilizes LLMs to accelerate development workflows. Our mission is to leverage retrieval augmented generation (RAG) to empower AI agents with project context. Equipped with relevant knowledge, agents can effectively aid in the creation, delegation, and completion of tasks as well as in answering project specific queries. We provide a unified application for development teams which seamlessly integrates into existing source controls and continuously updates to handle evolving software.

2. Dataset Information

2.1 Dataset Introduction

OTTO's dataset consists of software repository content from GitHub repositories and generated Q&A examples for fine tuning.

Primary Dataset: Source code repositories including files, commit history, branch structures, documentation, and metadata.

Purpose: To provide the RAG (Retrieval-Augmented Generation) system with contextual knowledge about software projects, enabling two core capabilities:

- Answering questions about the codebase (Information Service)
- Generating development tasks given product requirements (Task Service)

Relevance: Repository context is essential for both services. Without this grounding, the AI cannot provide project-specific answers or create relevant tasks.

Fine Tuning Dataset: Generated questions (unstructured text) and answers (unstructured text or structured JSON output depending on signal tokens).

Purpose: To generate a multi-modal model that responds differently to queries depending on the presence of signal tokens.

Relevance: This multi-modal nature is essential so that one model can be used for the Q&A service and the Task Service.

2.2 Data Card

Attribute	Details
Dataset Name	Software Repository Corpus
Size	Variable per repository; typical range of 10MB to 500MB of text content per project
Format	Raw source files (.py, .js, .ts, .java, etc.), Markdown (.md), JSON (package.json, configuration files), YAML
Data Types	Unstructured text (code, documentation) and semi-structured data (JSON configurations, commit metadata)

Attribute	Details
Dataset Name	Generated Q&A Interactions
Size	~ 500 input/output examples
Data Types	Unstructured text (queries and LLM responses) and structured data (JSON task responses)

2.3 Data Sources

Primary Source: GitHub API

Access Method: GitHub Authentication

Fine-Tuning Source: Team Created

2.4 Data Rights and Privacy

Strictly open source repositories will be used as data for this project. Within the application, users authorize access to their repositories via OAuth authentication. Otto only indexes repositories that the authenticated user has explicit permission to access. No public scraping of private code occurs without user consent. Commit history may contain personally identifiable information such as author names and email addresses. The system provides options to anonymize or exclude sensitive metadata. Data retention policies define how long indexed data is stored. Users maintain the right to deletion; disconnecting removes all associated repository data from the system. Embeddings and indexed content are stored with encryption at rest and in transit. Storage location and cloud provider region will be documented for compliance purposes.

3. Data Planning and Splits

The ingestion pipeline connects to repositories via the GitHub API, fetches file contents and directory structures, and extracts commit history, branch information, and pull request descriptions. Preprocessing involves parsing code into meaningful chunks organized by file, class, or function. Embeddings are generated for each chunk and stored with metadata including file path, programming language, and last modified timestamp.

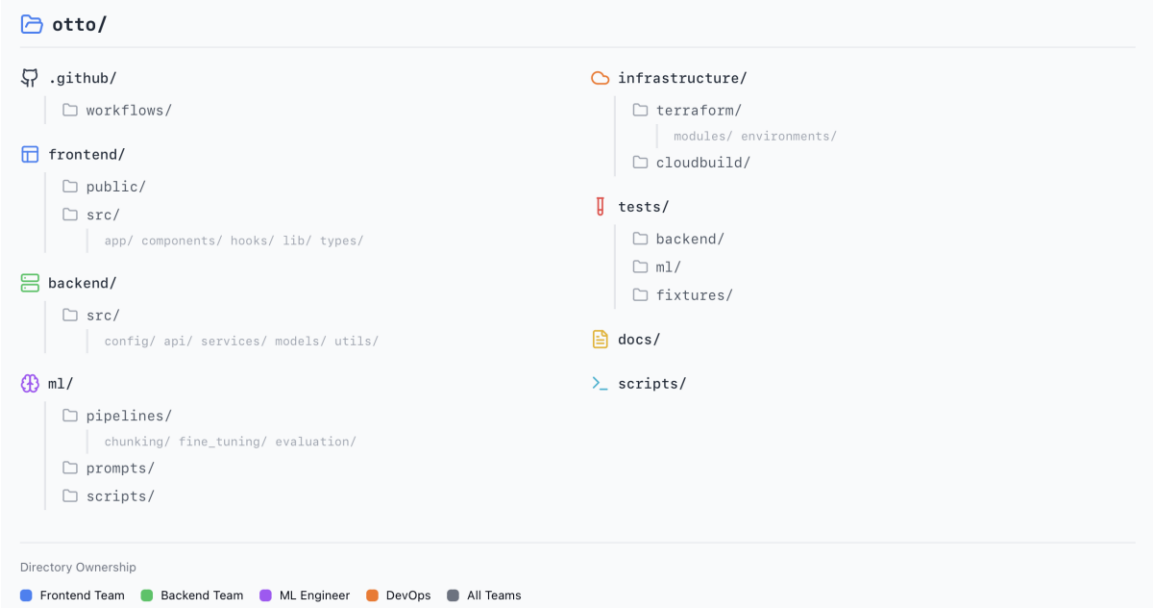
Since OTTO primarily uses RAG rather than traditional supervised learning, data splitting follows a different paradigm.

- For RAG retrieval, 100% of the indexed repository content is available for retrieval operations.
- For Task Service fine-tuning, formatted ticket examples are split into training (80%), validation (10%), and test (10%) sets.
- For Information Service fine-tuning, examples follow the same 80/10/10 split for training, validation, and testing.
- For evaluation purposes, held-out questions and tasks measure retrieval quality and output accuracy.
- All testing and production data will be stored in GCP.

4. GitHub Repository

Github Repository

4.1 Folder Structure



5. Project Scope

5.1 Problems

- (Task Creation) A project manager receives new product requirements and must manually create tasks for developers.
- (Project Specific Inquiry) An employee has a project specific inquiry that blocks them from working.

5.2 Current Solutions

- (Task Creation) Project managers must manually dissect product requirements into manageable tasks to assign to developers.
- (Project Specific Inquiry) Employees must either review comments in the code base, consult company documentation, or reach out to others to answer questions.

5.3 Proposed Solutions

- (Task Creation) Project managers can input new product requirements and tasks will automatically populate a Kanban board.
- (Project Specific Inquiry) Employees can ask questions via a chat interface and receive contextually relevant, accurate responses.

6. Current Approach Flow Chart and Bottleneck Detection

7. Metrics, Objectives, and Business Goals

7.1 ML Metrics

- **Context precision**, **context recall**, and **answer relevance** through RAGAS.

7.2 Business Metrics

- Improve **sprint velocity** of agile teams.
- Reduce **time spent** by project managers during sprint planning.
- Reduce **resolution time** for employee questions.
- Reduce **onboarding time** for software developers and project managers.

7.3 Objectives and Business Goals

- Develop a chatbot using RAG on the repository to help answer project specific questions asked by employees.
- Develop a task management agent that can create and modify tasks in a Kanban board.
- Integrate with source control platforms to allow for the migration of existing projects.

8. Failure Analysis

8.1 Development Risks

Timeline: Collecting 500+ quality training examples may prove difficult.

- **Mitigation:** Prioritize Info Service first to validate RAG approach, set weekly checkpoints to cut features if needed, and parallelize workstreams.

Model Quality: Fine-tuned models may underperform on edge cases or unfamiliar project types.

- **Mitigation:** Keep humans in the loop during testing, build automatic fallback to base models, and create benchmarks before committing to any model.

8.2 Infrastructure Risks

Compute: Vertex AI quota limits, cold starts, and zone outages could affect availability.

- **Mitigation:** Queue requests during spikes, keep services warm with periodic pings, cache responses as fallbacks.

Database: PostgreSQL failovers, connection exhaustion, and vector DB corruption or slowdown as it grows.

- **Mitigation:** Use connection pooling, take regular snapshots, plan for sharding if needed.

8.3 Pipeline Risks

RAG Indexing: Webhook timeouts, Cloud Function memory limits, GPU quota exhaustion.

- **Mitigation:** Automatic retries with backoff, chunked processing, CPU fallback for embeddings.

Model Deployment: New models may fail on real-world edge cases, rollback systems may not work.

- **Mitigation:** Test on peers before production, require statistical significance for promotion, maintain one-click rollback.

8.4 Launch Risks

Integrations: OAuth token expiration, API rate limits.

- **Mitigation:** Automate token refresh, batch and cache requests.

Operations: Model drift over time, vector DB slowdown, accidental sensitive data exposure.

- **Mitigation:** Monitor drift and retrain regularly, plan for index partitioning, auto-redact sensitive logs.

8.5 Risk Summary

Priority	Risk	Likelihood	Impact
Critical	Security or data leakage	Low	High
Critical	RAG pipeline failures	Medium	High
High	Running behind schedule	High	Medium
High	Hitting cloud quota limits	Medium	Medium
High	Not enough training data	Medium	High
Medium	GitHub integration issues	Low	Medium
Medium	Model accuracy degrading over time	High	Medium

9. Deployment Infrastructure

9.1 Core Architecture

Cloud Provider: Google Cloud Platform (GCP)

Layer	Components
Frontend	Next.js/React on Firebase Hosting or Cloud Run
Backend	FastAPI microservices (Task, Info, Integration) on GKE, Cloud Endpoints for API gateway
ML/AI	Vertex AI for model serving (fine-tuned Gemini 1.5 Pro with LoRA adapter), Vector Search (ScaNN) for RAG, Vertex AI Embeddings API (text-embedding-004), Vertex AI Model Registry for versioned adapters, Cloud Storage for processed chunks and training data
Data	Firestore (NoSQL), Cloud Storage (repository data, training datasets), Cloud Pub/Sub (webhook events)
MLOps	GitHub webhooks → Cloud Functions for indexing, Vertex AI Model Registry, Cloud Build for CI/CD
Security	Firebase Auth (OAuth 2.0/JWT), Cloud IAM, Secret Manager, Cloud Armor (DDoS protection)

9.2 Data Flow Patterns

1. API Requests: User → CDN → Cloud Endpoints (auth) → Microservice → Cloud SQL/Redis → Response
2. Task Generation: User requirement → Task Service → Vertex AI → JSON tasks → Cloud SQL
3. RAG Queries: User Query → Vertex AI Embeddings API → Vector Search (ScaNN) context retrieval → OTTO Model (fine-tuned Gemini with LoRA) → Response (Q&A text or Task JSON based on mode token)
4. Fine-Tuning Pipeline: Q&A Interactions (Firestore) → Export Training Data → Format as JSON with mode tokens ([QANDA], [TASKGEN]) → Vertex AI Supervised Tuning → LoRA Adapter → Model Registry
5. RAG Indexing: GitHub API → Cloud Storage → Sensitive Data Cleaning → Semantic Chunking (tree-sitter via Apache Beam/Dataflow) → Vertex AI Embeddings API → Vector Search Index

9.3 CI/CD Pipeline

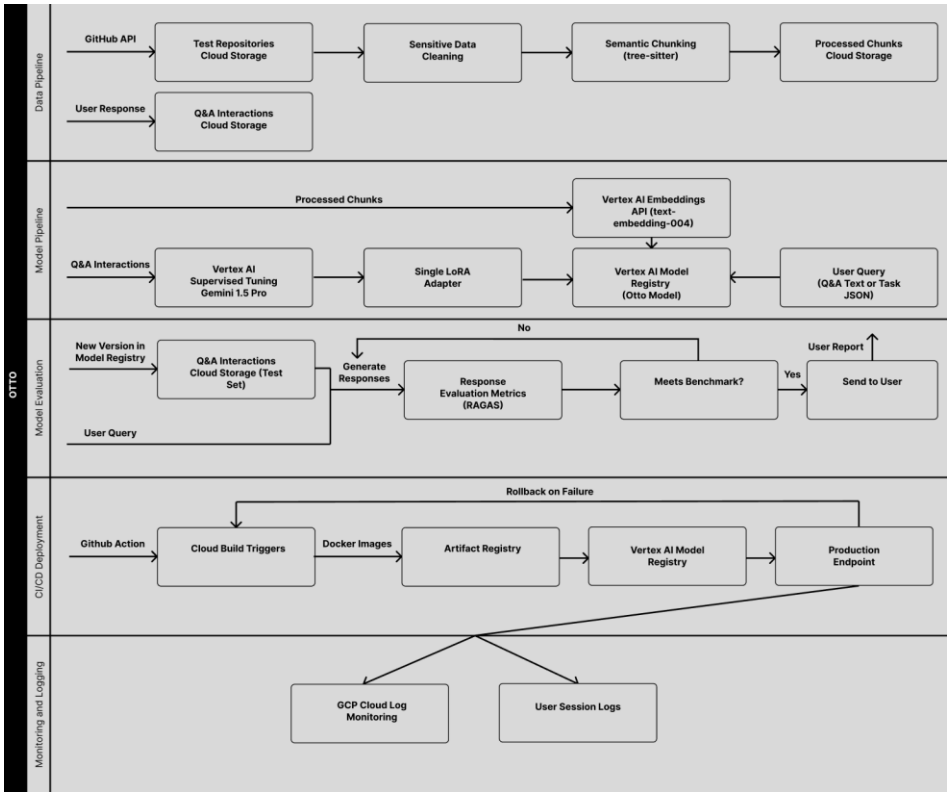
Application: GitHub push → Input Updated Repository Data to Vector DB namespace

ML Model: New training data threshold reached → Export from Firestore → Vertex AI Supervised Tuning (Gemini 1.5 Pro) → LoRA Adapter generated → RAGAS evaluation against test Vector Search index → Model Registry versioning → A/B shadow deployment → Gradual production rollout (25% → 50% → 100%) → Rollback on failure

9.4 Disaster Recovery

Aspect	Strategy
Backups	Daily Cloud SQL backups, Cloud Storage versioning, daily vector DB snapshots, Terraform IaC in Git
Availability	Multi-zonal GKE (99.95% SLA), Cloud SQL HA, active-passive multi-region
Recovery Objectives	RTO <2 hours, RPO <30 minutes

9.5 Model Deployment Diagram



10. Monitoring Plan

10.1 Objectives

Detect issues within 5 minutes, resolve critical problems within 30 minutes, prevent capacity-related outages, catch model accuracy drops >5%.

10.2 Tools

- GCP-native: Cloud Logging, Cloud Monitoring, Cloud Trace, Cloud Error Reporting, Vertex AI Model Monitoring

10.3 What We Monitor

Layer	Key Metrics	Alert Thresholds
Infrastructure	CPU, memory, pod restarts, DB connections, cache hit rate, queue backlog	CPU >80%, memory >85%, connections >80%, cache hits <70%
Application	Uptime, response time, error rate	<99.9% uptime, p99 >3s, errors >0.1%
ML/AI	Model drift, RAG relevance, LLM latency, user feedback	Relevance <0.75, latency >10s, positive feedback <80%
Security	Failed logins, unusual locations, API anomalies	>5 failures/5min, usage >3x baseline

10.4 Alerting

Severity	Examples
P1 Critical	Service completely down, risk of data loss
P2 High	Major performance degradation
P3 Medium	Elevated error rates, minor performance issues
P4 Low	Warnings, capacity planning items

10.5 Rollout

Week	Deliverables
7	Cloud Monitoring, infrastructure dashboards, P1 alerts
8	Structured logging, Cloud Trace, service dashboards, P2/P3 alerts
9	Model monitoring, RAG dashboard, user feedback collection, drift alerts
10+	Threshold tuning, usage dashboards, runbook documentation

11. Success and Acceptance Criteria

11.1 Functional Completeness

- All core services (Task, Info) are operational and integrated
- Project management interface is fully functional with GitHub integration
- RAG pipeline successfully indexes and retrieves from connected repositories
- End-to-end workflow demonstrated: user query → AI response → actionable output

11.2 Performance Benchmarks

- Info Service: Responds to repository questions with ≥85% relevance accuracy
- Task Service: Generates properly structured JSON tickets that pass schema validation 100% of the time
- Response latency: <5 seconds for standard queries, <15 seconds for complex task generation

12. Timeline Planning

The project follows a 12-week timeline divided into four phases, each with specific deliverables and milestones.

Phase 1: Infrastructure & Data Foundation (Weeks 1-3)

Week 1 — Set up GCP project with IAM roles, configure Firebase Authentication with GitHub OAuth, deploy Firestore and Cloud Storage buckets, initialize Terraform modules

Week 2 — Deploy FastAPI backend to Cloud Run, configure Cloud Endpoints API gateway, build GitHub App for webhooks, set up Cloud Pub/Sub, configure Cloud Logging

Week 3 — Deploy Vertex AI Vector Search (ScaNN), build Apache Beam chunking pipeline with tree-sitter, configure Vertex AI Embeddings API, run initial RAG experiments

Phase 2: Model Development (Weeks 4-6)

Week 4 — Deploy Q&A Agent service, configure Vertex AI Gemini 1.5 Pro endpoint, build chat interface, implement feedback collection, collect 200+ training examples

Week 5 — Deploy Task Generation service, implement mode token routing, set up Vertex AI Model Registry, configure supervised fine-tuning pipeline, reach 500+ training example

Week 6 — Execute first LoRA fine-tuning run, deploy OTTO Model to staging, set up Cloud Build CI/CD, build Kanban board UI, complete end-to-end workflow testing

Phase 3: Production Readiness (Weeks 7-9)

Week 7 — Implement RAGAS evaluation metrics, build test Vector Search index, set up JSON schema validation, configure Cloud Monitoring dashboards and drift detection

Week 8 — Build continuous training pipeline, set up automated retraining triggers, implement A/B testing infrastructure, configure model rollback procedures

Week 9 — Complete load testing, configure Cloud Armor and Secret Manager, verify Vector Search updates on repository push, document incident response

Phase 4: Launch & Documentation (Weeks 10-12)

Week 10 — Production rollout, user testing, bug fixes, performance tuning

Week 11 — Complete all documentation, finalize model cards, prepare demo

Week 12 — Final presentation, live demo, project submission