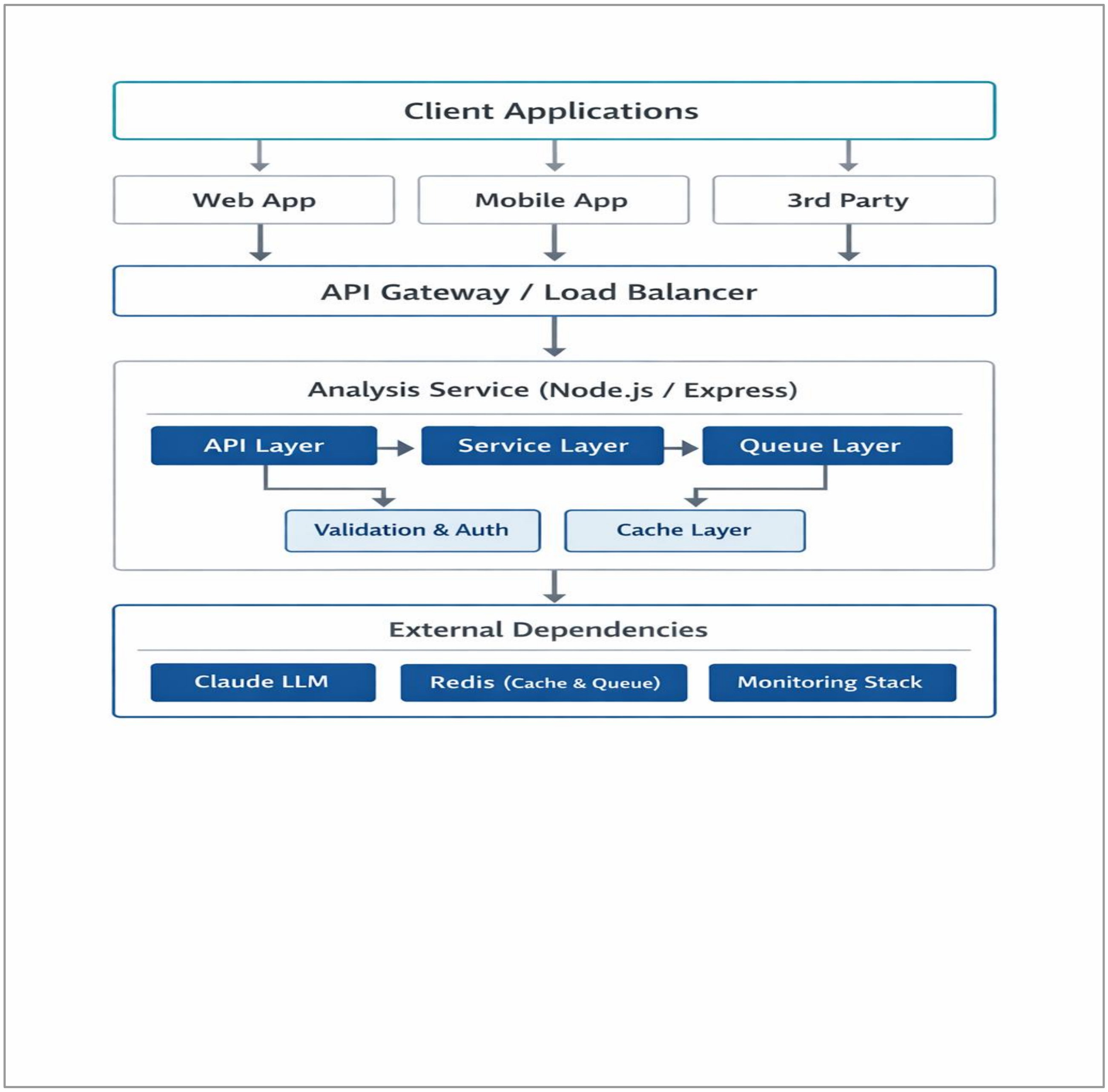
1. System Architecture

1.1High-Level Component Architecture



1.2. Data flow from request to response

Synchronous Analysis Flow:

1. Request Reception: Client sends POST request to `/api/analyze`

2. Validation: Input validation and sanitization

3. Cache Check: Check Redis for cached analysis using cache key or content hash

4. Cache Hit: Return cached response with cached: true flag
5. LLM Processing: If cache miss, call Claude API with structured prompt
6. Response Parsing: Parse and validate LLM response
7. Caching: Store result in Redis with TTL
8. Response: Return structured response to client

Asynchronous Analysis Flow (Alternative)

1. Job Creation: Client requests async processing
2. Validation: Same as synchronous flow
3. Queue Job: Add job to Bull queue with Redis backend
4. Background Processing: Worker processes job asynchronously
5. Result Storage: Store result in cache with job ID
6. Notification: Optionally send webhook or allow polling (Client polls GET /api/analyze/status/{job_id})

2. Key Design Decisions

2.1. Why this architecture?

Modular Monolith First Approach

- Single deployable unit for simplicity (reduces the complexity for initial launch)
- Clear separation of concerns within the codebase allows easy extraction to microservices later
- Easy to extract services later if needed
- All code in one repository simplifies tracing requests
- Small team can work effectively without microservices overhead

Node.js/Express: Fast prototyping, asynchronous support for LLM calls, team familiarity.

Redis Dual-Use: Caching to reduce LLM cost, queueing for async processing, pub/sub for real-time updates.

Provider Pattern: Abstracts LLM integrations, enables multi provider.

2.2 How responsibilities are separated

- **config/**: Configuration settings including redis.
- **controllers/**: Route handlers and business logic entry points.
- **middleware/**: Middleware for validation, error handling, authentication, etc.
- **providers/**: LLM provider abstractions and integrations.
- **routes/**: API route definitions.
- **services/**: Core business logic, orchestrating AI, caching, and queuing.
- **types/**: Type definitions for TypeScript, including interfaces and DTOs.
- **utils/**: Utility functions, logging, health checks, external service clients.
- **server.ts**: Application bootstrap and server initialization.
- **tests/unit/**: Unit tests for services (e.g., analysis.service.test.ts).

2.3 How AI logic is isolated from core business logic

1. Provider Pattern for LLM Abstraction : Abstract LLM interactions behind a common interface
2. Prompt Templates: Separated from business logic
   - Templates in prompt.service.ts
   - Easy to update without code changes
   - Support for multiple use cases
3. Response Validation: Independent validation of LLM outputs
   - Schema validation
   - Quality checks
   - Fallback generation

3. Scalability & Extensibility

3.1 How would this evolve with more use cases?

Current Architecture (Single Service)

POST /api/analyze

- Customer service analysis
- Delivery optimization
- Inventory insights
- Staff performance review

Phase 2: Specialized Endpoints (Service Split)

POST /api/analyze/customer-service

- Sentiment analysis
- Complaint categorization
- Resolution suggestions

POST /api/analyze/operations

- Process optimization
- Bottleneck identification
- Efficiency recommendations

POST /api/analyze/financial

- Cost analysis
- Revenue opportunities
- Risk assessment

Phase 3: Microservices Architecture

Scaling Strategies

Horizontal Scaling:

- Stateless application servers
- Load balancer with health checks
- Shared Redis cluster
- Distributed job queues

Database Growth:

- Currently: Redis only (ephemeral)
- Future: PostgreSQL for persistent storage
- Archival: Move old analyses to cold storage

Global Deployment:

- Multiple regions for reduced latency
- Regional LLM endpoints
- Geo-distributed caching

3.2 What Breaks First at Scale?

First Bottleneck: Claude API Rate Limits

Mitigation:

- Request queuing,
- multiple LLM proivders,
- Multiple API keys
- caching
- Add multi-provider fallback

Second Bottleneck: Redis Memory

Mitigation:

- LRU eviction,
- data partitioning
- Redis Cluster
- Set appropriate TTL values
- Monitor memory usage with alerts

Third Bottleneck: Node.js Event Loop

Mitigation:

- Worker threads (Offload CPU-intensive tasks to workers),
- horizontal scaling with load balancer ,
- Use async/await consistently,
- Monitor event loop lag
- Response caching, CDN

Fourth Bottleneck: Database Connections (Database Connection Pool Exhaustion)

Mitigation:

- connection pooling
- connection multiplexing
- Set appropriate pool sizes
- Implement retry logic with backoff
- Consider connection-less architectures
- Read Replicas

4. LLM-Specific Considerations

4.1 Prompt Strategy

Structured Prompts:

- Clear role definition
- Specific task instructions
- Output format specification
- Context and examples

Multi-Template Approach (Structured Prompt Templates)

Context Window Management

4.2 Handling Hallucinations & Bad Outputs

- Defense-in-Depth Validation (Multi-Layer Validation)

- Ensure JSON structure is correct
- LLM provides confidence estimate
- Human Review (Flag low-confidence outputs)

- Fallback Mechanisms

- Simplified analysis if complex one fails
- Multiple prompt attempts
- Rule-based fallback for critical failure

-Monitoring:

- Track hallucination rates
- User feedback mechanisms
- Regular quality audits

4.3 Evaluation or quality control ideas

- Enforce structured JSON output and validate required fields
- Reject or retry responses that don't match expected format
- Check for hallucinations by comparing output with input data
- Use confidence score returned by the model as a quality signal

- Track user feedback and corrections
- Version prompts and models to detect regressions
- Maintain a small set of test inputs for quality comparison
- Monitor retries, errors, and low-confidence outputs
- Clearly label outputs as AI-generated and assistive

Section 3 - Reliability, DevOps & Production Thinking

1. How Would You Deploy This Service?

I would deploy the service using one of the following approaches depending on scale and traffic patterns.

- **Containerized Deployment (Preferred)**

  The service is packaged as a Docker image and deployed to a Kubernetes cluster. Kubernetes runs multiple replicas of the Node.js service and handles scaling and restarts. A load balancer routes incoming traffic to the pods. Redis is used for caching and background jobs, either as a managed service or a StatefulSet. API keys and configuration values are injected using Kubernetes Secrets and environment variables.

- **Serverless Deployment (AWS Lambda)**

  API Gateway receives incoming requests and triggers a Lambda function. The Lambda function runs the analysis logic and calls the LLM. Secrets such as API keys are loaded from AWS SSM Parameter Store or Secrets Manager. Redis is accessed via Elasticache or an external Redis provider. This approach works well for spiky traffic and short-running analysis requests.

- **Traditional VM Deployment**

  The service is deployed on a virtual machine with Node.js installed. The application is run using PM2 for process management and auto-restart. NGINX acts as a reverse proxy and handles incoming traffic. Redis runs locally on the VM or is accessed as a managed service. Environment variables are configured using .env files or system services.

2. What would you monitor?

App Metrics

- API request success/failure rates
- Latency per request
- LLM response time
- Queue backlog

LLM-Specific Metrics

- Token usage (input/output)
- Model cost per request
- Frequency of errors/timeouts

System Metrics

- CPU/memory usage
- Autoscaling thresholds
- Rate-limit warnings

3. How would you handle

a. Timeouts / slow LLM responses

- Set timeout (10–15s)
- Retry with exponential backoff
- Fall back to partial summary if needed
- Use async queue for long requests

b. Partial failures

- Return structured error payload
- Include internal error codes
- Log failure cause with correlation IDs

- Continue processing unaffected components

c. Model regressions

- Implement model versioning
- Keep prompt templates versioned
- Add health-check comparison between versions
- Allow rollback via config flag

4. How would CI/CD fit in?

CI/CD Pipeline:

This CI/CD pipeline automates the full software delivery workflow from code changes to production deployment:

- **Developer Push:**
  A developer submits new code to the repository, triggering the pipeline.
- **GitHub/GitLab Webhook:**
  The version control platform detects the push and automatically notifies the CI system.
- **CI Pipeline (GitHub Actions):**
  The CI process begins, orchestrating all automated stages such as build, testing, and scanning.
- **Test Execution:**
  Unit tests and integration tests run to validate functionality and prevent regressions.
- **Build Docker Image:**
  The application is packaged into a Docker image to ensure consistent deployment across environments.
- **Security Scanning:**
  Tools like **Snyk** and **Trivy** analyze dependencies and the container image for vulnerabilities.
- **Staging Deployment (Canary):**
  A controlled rollout is performed in the staging environment to validate changes with real workload behavior.
- **Approval Gates:**
  Manual or automated approvals ensure only verified builds move forward.
- **Production Deployment (Blue-Green):**
  A new version is deployed alongside the existing one, allowing safe switching with zero downtime and instant rollback if issues appear.

Section 4: Planning, Estimation & Communication

1. If this were a real project:

a. How would you break it into milestones?

Project Timeline (8 Weeks)

Week 1-2: Foundation & Core MVP (Milestone 1)

- Project setup & basic API structure
- Claude API integration & prompt engineering
- Basic error handling & validation

Week 3-4: Production Readiness (Milestone 2)

- Caching layer with Redis
- Async processing with job queues
- Monitoring, logging & health checks
- Dockerization & deployment scripts

Week 5-6: Enhanced Features (Milestone 3)

- Multi-LLM provider support
- Advanced analytics dashboard
- Rate limiting & security enhancements

Week 7-8: Polish & Launch (Milestone 4)

- Comprehensive testing & bug fixes
- Documentation & training materials
- Staging deployment & user acceptance testin
- Production launch & monitoring

b. What would you ship first vs later?

Ship First

MVP Features

- POST /api/analyze
- Basic Claude integration
- Input validation
- Error handling
- Basic logging
- Health endpoint
- Basic prompt template

Ship Later

Phase 2: Production Features (Weeks 3-4)

Production Features

- Redis caching
- Async job processing
- Rate limiting
- Comprehensive monitoring
- Docker deployment
- API documentation

Phase 3: Enhanced Features (Weeks 5-6)

Enhanced Features

- Multi LLM provider support
- Advanced analytics dashboard
- Webhook notifications
- Batch processing

- User authentication

Phase 4: Future Roadmap

Future Features

- Real time streaming
- Advanced NLP capabilities
- Predictive analytics
- Integration with CRM systems
- Mobile app SDK

2. Rough time estimate for:

a. MVP

Component	Time (Days)
Project setup & scaffolding	1
Basic Express API structure	2
Claude API integration	3
Prompt engineering & testing	2
Input validation & error handling	2
Basic testing & documentation	2
Code review & refinements	2
Total	14 days

b. Production-ready v1

Component	Time (Days)
MVP (as above)	14
Redis caching implementation	3
Async job processing with Bull	4
Comprehensive monitoring & logging	3
Dockerization & deployment setup	2
Security enhancements	2
Performance optimization	3
End-to-end testing	3
Documentation & training	2
Total	36 Days

Note: (7 weeks with 1 developer)

Team Composition:

- 1 Senior Backend Engineer (full-time)
- 1 DevOps Engineer (part-time, 50%)

- 1 Product Manager (part-time, 25%)
- QA support as needed

3. Key risks you would flag to stakeholders

LLM Risks

- Hallucinations / inaccurate summaries may mislead staff.
- Model regressions when provider updates version.
- Token limit issues for long notes or large structured records.

Operational Risks

- High latency / timeouts depending on provider load.
- Unexpected downtime from external LLM provider.
- Scaling restrictions because LLM throughput is limited.

Cost Risks

- Token usage can grow very fast if prompts expand.
- High-volume processing becomes expensive without caching.
- Long notes or large JSON payloads inflate token cost.

Security & Compliance Risks

- Sensitive business/customer data sent to a 3rd-party API.
- Requires encryption, anonymization, and secure configuration.
- Must ensure compliance (GDPR, data residency rules, etc.).

Architectural Risks

- Vendor lock-in if the system is tightly coupled to one LLM.
- Need a pluggable adapter pattern to switch models in future.
- Bad prompt engineering could degrade output quality.

Evaluation Risks

- AI output quality is subjective and hard to measure.
- Requires a human-in-the-loop or automated scoring system.
- Need feedback loops for continuous improvement.